UNIVERSITY OF CAMBRIDGE

Guided Automatic Binary Parallelisation

Ruoyu (Kevin) Zhou

St John's College

This dissertation is submitted in July, 2017 for the degree of Doctor of Philosophy in Computer Science

Abstract

For decades, the software industry has amassed a vast repository of pre-compiled libraries and executables which are still valuable and actively in use. However, for a significant fraction of these binaries, most of the source code is absent or is written in old languages, making it practically impossible to recompile them for new generations of hardware. As the number of cores in chip multi-processors (CMPs) continue to scale, the performance of this legacy software becomes increasingly sub-optimal. Rewriting new optimised and parallel software would be a time-consuming and expensive task. Without source code, existing automatic performance enhancing and parallelisation techniques are not applicable for legacy software or parts of new applications linked with legacy libraries.

In this dissertation, three tools are presented to address the challenge of optimising legacy binaries. The first, GBR (Guided Binary Recompilation), is a tool that recompiles stripped application binaries without the need for the source code or relocation information. GBR performs static binary analysis to determine how recompilation should be undertaken, and produces a domain-specific hint program. This hint program is loaded and interpreted by the GBR dynamic runtime, which is built on top of the open-source dynamic binary translator, DynamoRIO. In this manner, complicated recompilation of the target binary is carried out to achieve optimised execution on a real system. The problem of limited dataflow and type information is addressed through cooperation between the hint program and JIT optimisation. The utility of GBR is demonstrated by software prefetch and vectorisation optimisations to achieve performance improvements compared to their original native execution.

The second tool is called BEEP (Binary Emulator for Estimating Parallelism), an extension to GBR for binary instrumentation. BEEP is used to identify potential thread-level parallelism through static binary analysis and binary instrumentation. BEEP performs preliminary static analysis on binaries and encodes all statically-undecided questions into a hint program. The hint program is interpreted by GBR so that on-demand binary instrumentation codes are inserted to answer the questions from runtime information. BEEP incorporates a few parallel cost models to evaluate identified parallelism under different parallelisation paradigms.

The third tool is named GABP (Guided Automatic Binary Parallelisation), an extension to GBR for parallelisation. GABP focuses on loops from sequential application binaries and automatically extracts thread-level parallelism from them on-the-fly, under the direction of the hint program, for efficient parallel execution. It employs a range of runtime schemes, such as thread-level speculation and synchronisation, to handle runtime data dependences. GABP achieves a geometric mean of speedup of 1.91x on binaries from SPEC CPU2006 on a real x86-64 eight-core system compared to native sequential execution. Performance is obtained for SPEC CPU2006 executables compiled from a variety of source languages and by different compilers.

Acknowledgements

First and foremost, I would like to thank my supervisor Dr. Timothy Jones who guided me throughout my years in Cambridge. I appreciate the opportunities and challenges he offered me, and his valuable support that made this dissertation possible. I would have never got this far without his expertise and assistance. No matter how tough the technical challenges I encountered, he is always there for discussion and encourages me to find solutions and carry on.
My sincere appreciation also goes out to:

- Edmund Grimley-Evans, my advisor at ARM, for his invaluable insight in many technical challenges I faced. During my placement at ARM, thanks to his enormous support, my understanding in dynamic binary translation has significantly strengthened.
- Robert Mullins, for providing constant source of encouragement and research ideas throughout my years in the Computer Lab from MPhil to PhD, especially during the weekend.
- Niall Murphy, for his assistance in the implementation of parallel execution models.
- Tom Sun, for his assistance in finding parallelism from SPEC CPU2006 and providing many legacy binaries compiled by different compilers.
- Sam Ainsworth, for his assistance in providing the software prefetching optimisation algorithms and test suites.
- Dennis Zhang, for his assistance in resolving a few significant technical challenges.
- Negar Miralaei, who tragically died shortly before finishing her dissertation, for her kind support and encouragement during the hard times.
- Everyone in Computer Architecture Group, for discussions and exchange at scrum meetings every two days.
- St John's College for funding my tuition fee during my study and research in Cambridge, for providing the best scenery, environment, food and accommodation in Cambridge.
- ARM Ltd for funding my tuition fee and maintenance during my study and research in Cambridge.
- My wife and parents, for always being been so loving, caring and supportive during my research.

Contents

Chapter 1

Introduction

The relationship between the computer architecture and software industries was described by Herb Sutter as "*Andy giveth, and Bill taketh away*"[1]. Before the early 2000s, the computer architecture industry delivered reliable and continuous processor performance enhancements through increased clock speeds and micro-architectural improvements in uniprocessors. No matter how fast processors got, software consistently has consumed the extra performance gain by adding more functionality. However, since the early 2000s, the computer architecture industry failed to continue to provide further improvement for uniprocessors. It is faced with diminishing returns in single-core performance at the cost of a significant increase in power, thermal dissipation and design complexity. As a result, computer architects have shifted development efforts to *chip multicore processors* (CMP) that place multiple cores in the same chip. These multicore processors can improve system throughput and improve performance for multiple processes and parallel applications.

However, the migration to CMP has presented a huge performance discontinuity for single-threaded software: latencies of single-threaded applications become even worse. Due to the power dissipation ceiling for multicore processors [2], cores are made simpler to meet the average power requirement. Each core is clocked at a lower frequency if all cores are switched on. What's worse, the average memory bandwidth for each core is also less than for an equivalent uniprocessor. As a result, the sequential performance on a single core in a CMP is undoubtedly lower than a uniprocessor with the same power budget.

Efficient utilisation of CMPs is becoming the dominating challenge for compiler research. For decades, software industries have stacked a vast repository of sequential programs and libraries that were single-threaded. Some of them are still valuable and actively in use. Most of their source code is either absent or protected, making it impossible to recompile on new generations of hardware. Even with the availability of source code, it would be much more expensive and time-consuming for programmers to write entirely new parallel programs.

1.1 Binary Recompilation

Every year new generations of hardware have constantly been released to the market. Currently, commodity desktop and server class processors from 8 to 16 cores are the norm. As more software is labelled "legacy" due to the advancement of hardware, it is important not to overlook the fact that sequential performance of more legacy software becomes increasingly sub-optimal.

One effective approach to improve legacy sequential software performance is to modify legacy machine code to adapt to newer generations of hardware and achieve better performance. Modifications of the machine code are also called binary *recompilation*. It refers to the process of transforming original binary instructions into more efficient forms while achieving the same program behaviour. Binary recompilation can be performed statically or dynamically through *binary translation*.

Static binary translation typically lifts the binary bytecode to an intermediate form using decompilation techniques. From the intermediate representation, it performs transformation and then compiles to a newer version of the binary. However, the machine code for legacy software is typically stripped and obfuscated. The lack of symbolic information, the mixture of data and code and the existence of indirect branches prevents an accurate static analysis. Therefore for complicated binaries, static binary recompilers typically fail to recover sufficient semantic information to apply optimisation while maintaining the same program output. On the other hand, dynamic binary translation modifies and executes the binary at the granularity of a basic block at a time. Each basic block is modified and immediately executed at runtime. Therefore, dynamic translation is more flexible and robust compared to static translation. However, dynamic approaches suffer runtime translation overhead. Modifications are also limited to a block at a time due to the lack of global understanding of the whole program.

A combination of static and dynamic binary translation together builds on each other's strengths. A static analysis of the binary can resolve the lack of global understanding of programs in dynamic translation. In turn, dynamic translation can access runtime information to alleviate the ambiguity. It also generates efficient just-in-time code to adapt to the underlying hardware components and achieves better performance.

In this dissertation, the first research hypothesis is that a combination of static binary analysis and dynamic binary translation can bring efficient binary recompilation and achieve better performance compared to the native execution of the original binary. To test the hypothesis, an open binary recompilation framework called GBR (Guided Binary Recompilation) is implemented. GBR performs static binary analysis to determine how recompilation should be undertaken, and produces a domain-specific hint program. This hint program is loaded and interpreted by the GBR dynamic binary translator. The problem of ambiguous static analysis is addressed through the cooperation between static hint program generation and just-in-time optimisation in dynamic binary translation. In this manner, complicated recompilation of the target binary is carried out to achieve optimised execution on a real system.

1.2 Automatic Parallelisation

The capability of comprehensive binary recompilation from GBR enables optimisation opportunities for automatic parallelisation of legacy software binaries. *Parallelisation* refers to the process of identifying (or creating) independence among program statements and scheduling them for concurrent execution across multiple cores to achieve performance. Typically, parallelisation of a sequential program is performed manually by modifying the original source code. Over the past decades, many manual parallel language APIs, directives and libraries have been developed, such as pthread[3], MPI [4], OpenMP [5] and Intel TBB[6].

Despite all this support, re-writing existing sequential programs to be parallel is difficult, even for skilled programmers. John Hennessy called parallelisation "a problem thats as hard as any that computer science has faced" [7]. Moreover, determining how to parallelise sequential applications optimally is a much more challenging task [8]. Programmers need to understand numerous details about the parallel algorithms, underlying hardware and spend tremendous efforts in verification and debugging. There are also new potential bugs introduced by parallelism, such as race conditions that are never there in a sequential program.

To relieve programmers from manual parallelisation, *Automatic Parallelisation* techniques have been introduced within tools such as compilers. Automatic parallelisation extracts parallelism from sequential programs and translates them into parallel versions. Rather than investing time and effort in parallel programming, programmers can leave the tool to do the parallelisation and pay more attention to developing more innovative applications. Recent automatic parallelisation approaches (HELIX [9], DSWP [10], SUIF [11], Pluto [12], Polly [13]) have demonstrated performance improvements over general purpose or scientific sequential programs. However, all of the techniques require the original source code for analysis. For legacy machine executables or libraries whose source code is protected or absent, automatic parallelisation techniques are not applicable.

The second research hypothesis is that existing automatic parallelisation transformations which were developed for compilers' intermediate representations can be also applied to application binaries through binary recompilation with the constraints of limited symbolic information without source code. Overall program speedup can be achieved through automatic binary parallelisation on a real system, compared to the native sequential execution of the binaries. To test the hypothesis, firstly the potential of thread-level parallelism is identified through static binary analysis and binary instrumentation in the GBR. The binary instrumentation framework is called BEEP, (Binary Emulator for Estimating Parallelism) that drives dynamic parallel cost models to estimate executable speedup. Secondly, once it proves enough parallelism, the automatic parallelisation transformation is implemented in GABP (Guided Automatic Binary Parallelisation). It combines the techniques from automatic parallelisation and binary recompilation and overcomes the limited data and control flow information using static analysis, profiling and just-in-time compilation.

1.3 Contributions

This dissertation aims to address the problem of CMP utilisation for legacy software binaries by optimising performance through dynamic binary recompilation and specifically recompiling binaries to enable automatic parallelisation. It is related to techniques in mature fields of research such as binary analysis, binary translation, runtime code optimisation and automatic parallelisation. Many systems with different goals and designs have utilised these technologies, including compilers, emulators, simulators, virtual machines, dynamic optimisers, and dynamic translators. However, recompiling binaries for parallelisation remains a fundamental blank space that has not been addressed by other systems.

Therefore, the original contributions of this dissertation are as follows:

- **Guided Binary Recompilation**: This dissertation proposes a novel static-dynamic approach for binary recompilation. It performs static binary analysis and generates domain-specific hint programs to automate the fine-grained modification process of dynamic binary translation. Compared to other binary translation framework, it combines the strength of both static binary analysis, dynamic binary translation and just-in-time compilation and overcomes the weaknesses of each. The prime novelty can be summarised as:

  – It proposes a standardised interface that conveys information from static binary analysis to dynamic binary translation.
  – It simplifies and decomposes the binary recompilation problem by treating the static hint generation as a *compilation process* and the dynamic binary translator as a *virtual machine* to interpret the static hint program.

  The proposed guided binary recompilation can achieve speedup on a real system compared to native execution of the original binary. The utility of GBR is demonstrated by adding software prefetch instructions to binary applications and vectorising parts of them to achieve performance improvements.

- **Guided Binary Instrumentation for Parallelism Analysis**: This dissertation proposes a framework that is used to identify potential thread-level parallelism through the hybrid of static binary analysis and guided binary instrumentation. Compared to other limit studies in parallelism, the prime novelty can be summarised as:

  – It performs preliminary static analysis on binaries and generates questions that are encoded in static hints. The questions are then answered by the dynamic binary translator. It is called *Demand-driven Instrumentation.*
  – It implements parallel execution models that combine the information delivered from static binary analysis and the runtime information and evaluates the potential parallelism based on existing parallelisation paradigms.

- **Guided Automatic Binary Parallelisation**: The dissertation proposes an automatic binary paralleliser that selects loops from sequential application binaries and automatically extracts thread-level parallelism from them on-the-fly, under the direction of the hint program, for efficient parallel execution. From the results, it can achieve performance speedup on a real system for SPEC CPU2006 benchmark executables. The prime novelty can be summarised as:

  – It minimises threading overhead by using static binary analysis to guide just-in-time compilation in the dynamic binary translator.
  – It minimises thread privatisation overhead by generating **multiple** different thread-private copies of loop code from the same original binary loop code.
  – It implements a novel software transactional memory called `JITSTM` whose entire code is dynamically generated. The `JITSTM` is used to support thread-level-speculation-based parallelisation.
  – It implements a hybrid of parallelisation schemes that handle data dependencies under the control of the hint program from the static binary analysis and profiling information. The hint program controls the dynamic code generation

that enforces runtime dependencies using thread-level speculation, synchronisation or value prediction.

1.4 Structure of this Dissertation

The structure of the dissertation is arranged as follows:

- Chapter 2: An overview of related background knowledge on dynamic binary translation, static binary analysis and automatic parallelisation techniques.
- Chapter 3: Describes the first contribution: guided binary recompilation. Details of the guided binary recompilation are presented. The chapter also lists two case studies of automatic software prefetching and vectorisation to demonstrate the effectiveness of GBR
- Chapter 4: Describes the second contribution. The chapter gives a comprehensive limit study in extracting various forms of parallelism from executables compiled from major benchmarks. The study also suggests feasible approaches for further parallelisation implementation in chapter 5.
- Chapter 5: Describes the third contribution. The chapter gives a description of the fundamental mechanisms of the GABP framework.
- Chapter 6: An evaluation of the performance by applying automatic parallelisation on a real system.
- Chapter 7: A conclusion and summary of future work.

Chapter 2

Background

The chapter presents related knowledge and design choices made to build a binary recompilation and parallelisation infrastructure. To recompile and parallelise a complicated executable, we need a system with three substantial sets of features. Firstly, the system needs to correctly *analyse* the given executable. It should identify possible regions that could be scheduled for recompilation. Secondly, the system needs to *recompile* the executable to the desired and optimised form. It should respect the original dependencies to preserve the same observable program behaviour. Thirdly, the system can extract task-level parallelism from the original binary to be scheduled for parallel execution. From the given three types of features, I discuss the related work from three fields in the following order: *dynamic binary translation*, *static binary analysis* and *automatic parallelisation.*

2.1 Dynamic Binary Translation

The foundation of recompiling and modifying binaries is the *binary translation.* Binary translation refers to converting source machine code compiled for a platform (source architecture) to another platform (target architecture). The translated target machine code could be in the same instruction set as the source machine code, which provides opportunities for optimisation, debugging and testing features. As binary translation can be performed across different instruction sets, it resolves compatibility issues and enables source application to be executed on incompatible machines.

Binary translation can be either performed statically or dynamically for the target platform. However, the static translation is never a complete solution [14]. Typically, binary executables are compiled on von Neumann machines [1]. There exist a large number of ambiguities due to the specification of the hardware architecture, where indirect control flows, mixed layout of data and code sections, dynamic linking and self-modification features are common. With the ambiguities, static translation fails to retrieve enough information for correct translation for complicated executables.

The lack of the symbolic and control information can be partially compensated by extra debugging, profiling and linking meta-data. Some static binary translators such as Peephole Superoptimizers [15], Etch [16], DIABLO [17], ATOM [18] use the extra information to compensate for the impact of the loss of information and ambiguities. However, the extra information is not available for typical legacy binaries. Hence, the

[1]von Neumann architecture: code and data reside in the same memory

applicability of static binary translators is limited. Static translators are rarely used by mainstream binary translation users.

On the other hand, dynamic binary translation (DBT) overcomes the lack of information problem. DBT translates the binary as the application is running natively. DBT is similar to binary emulation, where it interprets each source instruction using different instructions on the target machine. Emulation is relatively slow because the emulation process is typically an expansion of code. The emulated resource of each instruction is de-allocated after finishing the instruction. DBT optimises the emulation by buffering the emulated instructions in another allocated memory region (code cache) so that the interpreted code might be revisited again without duplicated interpretation.

There is a substantial body of DBT work for system virtualisation, such as VMware [19] or Virtual PC [20]. Here we only focus on user-mode binary translation on a single application. User-based DBT systems operate directly on program binaries with no need to access the source code of the guest application. It allows users to analyse and manipulate legacy code, proprietary code and streaming code in a straightforward and robust manner. Users may execute the whole program under the control of DBT systems or attach DBT to already running processes.

The flexibility of DBT enables many opportunities for program analysis, binary modification and performance optimisation. In this project, I use DBT as an efficient platform to implement a *Just-In-Time binary recompiler* that aims for performance. Therefore, only performance-oriented DBT features are discussed more in detail.

2.1.1 Translation Process

In general, dynamic binary translators follow three different types, based on their styles of execution:

- **Interpretation** style: the original binary is viewed as data. Each original host instruction is a callback to a piece of code that achieves the same functionality on the target machine.
- **Probe** style: the original binary is patched in-place by overwriting the original instructions with jumps or interrupts to the new code.
- **Just-In-Time** style: It translates each basic block before execution. The block is translated and buffered in the code cache. It then runs directly on modified code in the code cache.

The Interpretation-based DBT is widely used in fast binary emulators. Despite its re-use optimisations, it is still slow as interpreting each instruction results in code expansion. Probe-based DBTs can only be used when translating on the same compatible hardware. The translation is limited when modifying CISC instructions with variable lengths.

Therefore we focus on DBTs that follow the JIT style. A JIT DBT reads the runtime instruction stream one block at a time, translates each block of code and copies the translated code into a code cache. The translation process runs in units of *basic blocks*. A basic block is defined as a sequence of instructions that ends with a single control transfer instruction (jump, call, etc.). Once a block is copied to the code cache, the DBT executes the translated block natively on the target architecture.

The following code shows the main execution loop of dynamic binary translation of JIT style.

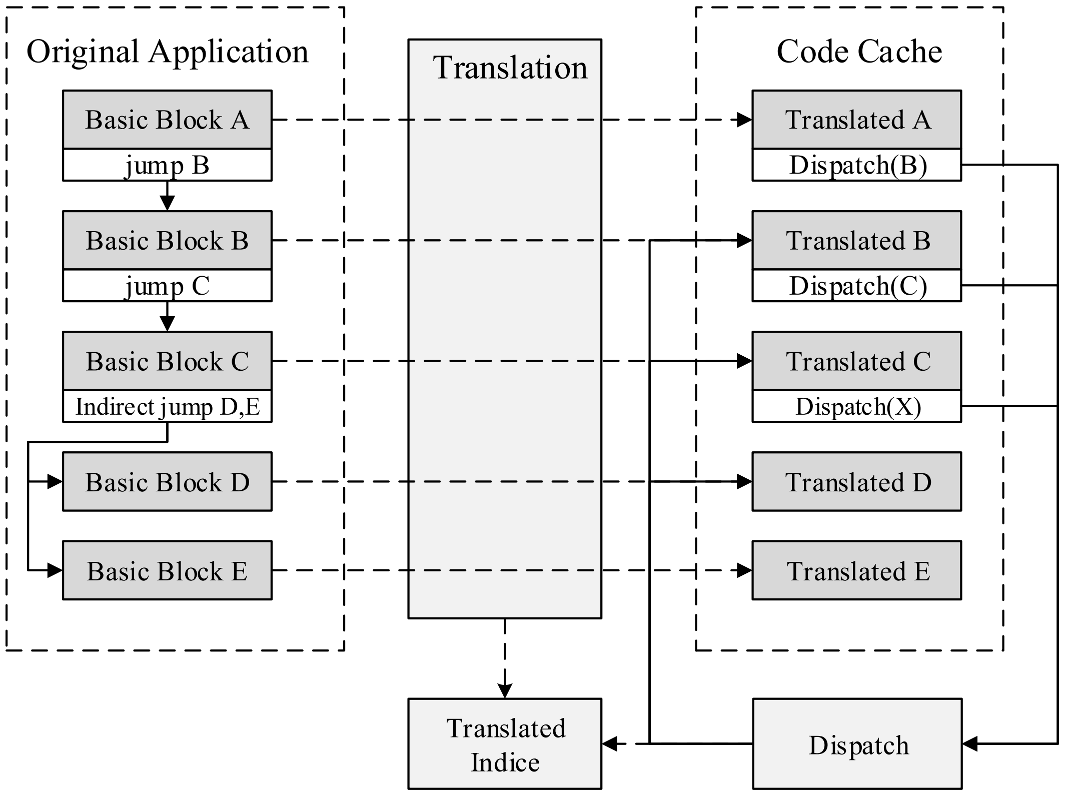


Figure 2.1: Dynamic Binary Translation

```
while (true) {
dispatch:
    if (!(translated_pc = find_translation(pc))
        translated_pc = translate(pc);
execute:
    pc = execute(translated_pc);
}
```

The last control transfer instruction of the block is replaced with a branch to the DBT *dispatch* procedure. The dispatch refers to the process of determining the next block to translate or execute in DBT. As shown in the above code, the dispatch procedure includes a search to find out whether the next incoming block has been translated. If the block is already translated, the control is redirected to the translated code in the code cache. And it only invokes translation when the incoming block is not found from its code cache. As new blocks of the target application are translated, the code cache is incrementally populated until eventually, the application runs entirely within the cached copy. Therefore DBTs typically suffer significant code cache warmup overheads if basic blocks are executed infrequently by the underlying application.

2.1.2 Just-In-Time Recompilation

When the DBT translates and copies a basic block to its code caches, it may perform modifications on the block for its purposes. The modified block is buffered and immedi-

Original basic block

```
add r0, r1
cmp [r0], r2
jle target
```

Translated basic block (Pseudo Code)

```
struct cpu {
    uint64_t r0;
    uint64_t r1;
    uint64_t r2;
   ...
};
s.r0=s.r0+s.r1
if (*(s.r0))<=s.r2)
    goto translate(target)
```

Figure 2.2: Binary translation by CPU virtualisation

ately executed in the code cache. We call the process *Just-In-Time Recompilation*. If the host and target machines share the same ISA, the fastest approach is to simply copy the original byte codes if they do not require modification. However, we still need to recognise and modify the last control transfer instruction of a block from its byte code to maintain control of the translation process. If explicit modifications are required, the block byte code is decoded into a list of internal, low-level *intermediate representations* (IR). The instructions in IR form facilitate platform-independent and easier modifications. After transformation, the IR is encoded back to byte codes. This method achieves the least translation overhead and therefore it is adopted by several high-performance DBTs such as DynamoRIO [21], the PIN tool[22] and MAMBO [23].

However, performing individual modification on each block is sub-optimal. When performing dynamic changes, the DBT is short-sighted and lacks a global view of the whole application. It typically steals a fraction of machine resources and inlines the modified code in the original basic block. Since it shares the same machine context as the original application, it is very expensive to request more registers to perform large-scale modifications. If more resources for modifications are required, the DBT typically performs a full context switch from the application.

If the DBT is not aiming for performance, there exists an alternative approach for modifications. Instead of stealing machine resources, it simulates the original machine states by allocating a dynamic structure of CPU register files and system registers. All operands from original instructions are translated to accessing the simulated CPU architecture. The process is a form of *virtualisation*. It is widely used for translations across different architectures and heavy-weight binary analysis tools such as Valgrind[24] and QEMU [25].

Similarly, it converts all basic block byte codes into IRs while adding one extra level of indirection through the virtualised structure. For example, Figure 2.2 shows that a basic block from original machine code is translated based on the simulated structure `struct cpu` (in pseudo code). Each register access of `r0` and `r1` transforms into accessing the corresponding element from the structure. By correctly simulating the semantics of each instruction, the correct state of the machine context is maintained. As all machine contents are simulated in memory, the registers on target machines become available for executing newly recompiled code. With more freedoms in using target machine registers, the virtualised approach can be used for more comprehensive binary modifications. After

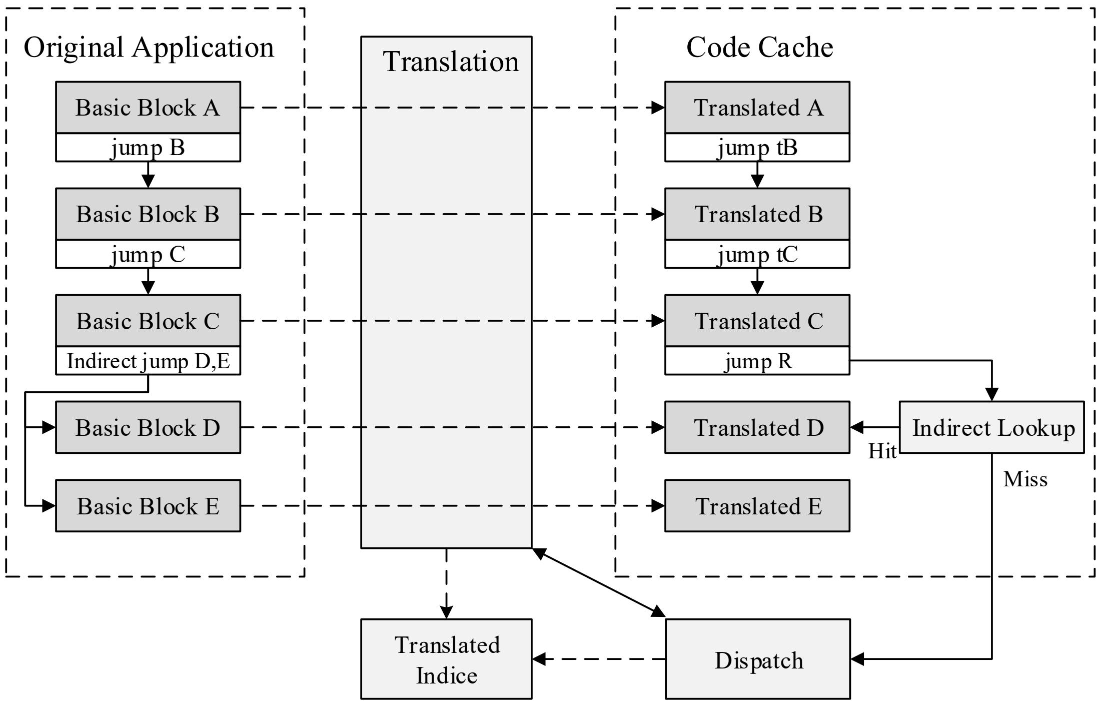


Figure 2.3: Fragment linking and indirect branch lookup

translation, the DBT performs optimisations using state-of-the-art compiler optimisation techniques, such as constant propagation, common subexpression elimination, etc. The final IR is compiled to byte codes on the target machine.

However, storing machine contexts in memory adds an extra indirection on top of the original semantics of the program, making it difficult to recognise opportunities for optimisation. There are also techniques [26] that map elements from the simulated structure to the original target machine registers for further performance improvement. Despite the optimisation efforts, virtualisation is still much slower than the native execution of the original binary. DBT that follows the virtualisation approach (e.g., QEMU) is around ten times slower than the light weight DBT such as DynamioRIO [27].

2.1.3 Linking and Indirect Branch Handling

As the translation process is an expensive task, it is vital to reuse the code as much as possible. If the target of a block is already translated and buffered, and it is targeted to another translated block via a direct branch, the two blocks can be directly linked together. The *linking* is implemented by directly overwriting the ending branches with new targets pointing to another translated code. Therefore, the DBT is a type of *self-modifying* application, due to its replacement of branch targets in during the link procedure.

However, indirect branches cannot be linked in the same way as direct branches because their targets may vary. Usually, a DBT performs a search from the translation table to determine the translated target of the dynamic instance of the indirect branch. If the indirect branch is in a frequently executed routine (e.g., in a loop), the overhead for lookup might be even more expensive. One optimisation is to inline the hash table lookup into the original instruction stream [28] without performing context switches. As there are different types of indirect branches such as indirect jump, function calls and returns, several hash table could be implemented to reduce collision rates of the hash table lookup.

2.1.4 Trace Optimisation

Even with inlined indirect branch lookup routine, a single indirect branch is expanded into tens of dynamic instructions for hash table lookup. The overhead for finding the address of the next translated block for indirect branches is still significant [29]. A possible optimisation is to *speculate* on frequently taken targets of indirect branches. For example in Figure 2.5, basic block `C` has an indirect branch targeting to either block `D` or `E`. If the block `D` is more likely to be taken, the code of `D` is appended to the previous block C. And one extra runtime validation is inserted to check whether the actual branch target is the same as predicted. If the prediction is wrong, it falls back to normal indirect branch lookup.

Branches from multiple basic blocks are removed and the remaining code is stitched together, forming the concept of *traces* or *superblocks*. Figure 2.5 illustrates the formation of traces. Direct branches such as A-B, B-C are removed, and basic block A,B and C are rearranged into a superblock. For indirect branch C-D,E, we speculate that the more frequent branch target is D and therefore we append the block D into the trace. There is an extra check to validate whether the actual target stays on the trace. If mis-speculated, it would fall back to normal indirect branch lookup routines.

Predicting the target of next indirect branch is called *speculative chaining* of basic blocks [31]. The concept of speculating on branch targets at runtime is reminiscent of the branch prediction in processors. Many well-known DBTs perform different chaining strategies: Dynamo [32] uses lightweight runtime profiling information to identify frequent targets for indirect branches. Pin [22] attempts to reduce overhead by jumping to a candidate destination block that begins with a compare to determine whether it is the correct destination, and on failure branches to the next candidate.

In addition to indirect branch optimisation, traces can also provide better code layout by removing direct branches and inlining of function calls. With trace optimisation, DBT could achieve better performance beyond the native execution for some applications [32, 33].

2.1.5 Dynamic Binary Instrumentation

With the JIT organisation of modify-copy-execution in dynamic binary translation, it is relatively easy to insert callbacks during the translation stage. By inserting trampoline calls to pre-compiled analysis code, we can examine runtime contexts and study dynamic behaviours of binaries. This process refers to *dynamic binary instrumentation* (DBI). For example in Figure 2.6, the function `memory_monitor` is a pre-compiled code snippet which is not from the original application. The function call is inserted and invoked before the actual memory access [`rax`]. The dynamic memory address is recorded and analysed in the function `memory_monitor`.

The inserted code for DBI such as `memory_monitor` should follow the *transparent* principle. A transparent DBI requires three major conditions. Firstly, the changes made by inserted instrumentation code should not be observed by the original application. From the example, the transparency is achieved by inserting additional spill/recover code to maintain the live machine states. Secondly, various runtime attributes collected from

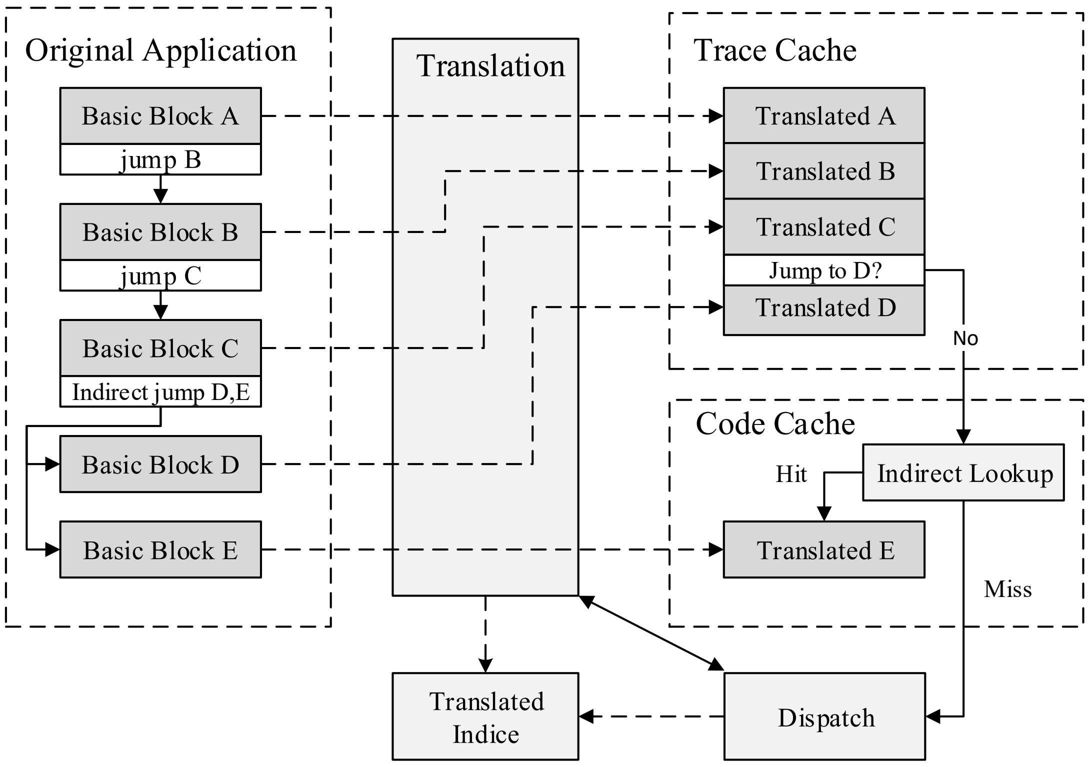


Figure 2.4: Building traces from basic blocks

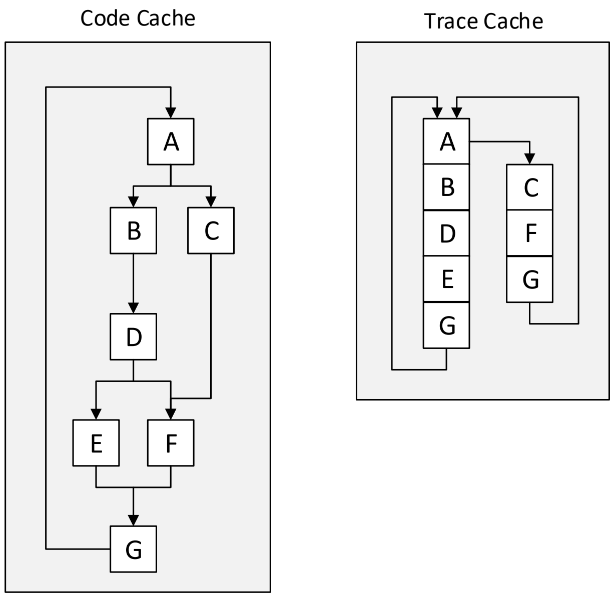


Figure 2.5: Overview of the Next Executing Tail (NET) trace creation [30]

Original basic block

```
add rax, rbx
cmp [rax], rcx
jle target
```

Modified basic block

```
add rax, rbx
spill caller-save-regs
lea rdi, [rax]
call memory_monitor
restore caller-save-regs
cmp [rax], rcx
jle target
```

Figure 2.6: Example: modification to enable instrumentation of the memory access `[rax]`, assuming the original binary uses the same calling convention as the instrumentation code.

instrumentation code should represent the exact runtime state when running without instrumented code. It requires the execution of the extra instrumentation code to meet the requirement of I/O or real-time conditions. Thirdly, code or libraries used in the instrumentation code should be re-entrant and isolated from the original application. For example, suppose there exists `malloc` functions from the original application, the inserted instrumentation code should not use the same `malloc` function when instrumentating memory accesses of the `malloc` function.

With transparent instrumentation, it is feasible to perform dynamic binary analysis to gain insight of applications at various points in execution. It highlights one of the fundamental differences between static binary analysis and dynamic binary analysis. Rather than considering what may occur in the static analysis, dynamic binary analysis operates on runtime events during the execution with no ambiguity. However, the results of the dynamic analysis are dependent on the input of the executable. Different combinations of inputs are required for more comprehensive dynamic analysis.

Transparent DBI have made it possible to develop some advanced dynamic binary analysis tools, for example, profiling and coverage analysis [34]; memory analysis [35, 36]; security analysis [37] and data race checking[38].

2.2 Static Binary Analysis

The second field of related work is *static binary analysis*, which is a subset of *static program analysis*. Static program analysis is a process of reasoning about the property of a program without actual running the program. It considers all possible different combination of inputs in contrast to dynamic program analysis. Typically static program analysis is performed with source code as inputs, while static binary analysis directly analyses binary objects. Both can abstract their inputs to the same data structure: the *program dependence graph* (PDG) [39]. They typically apply the same analysis algorithm from the PDG.

There are many usages for static binary analysis such as debugging, verification, reverse engineering, security analysis, etc. This dissertation only focuses on analysis techniques specifically for automatic recompilation and parallelisation. In this section, I will discuss generic techniques to process binary executables, construct a PDG and the analysis of control and data flows in the PDG.

2.2.1 Binary Abstraction

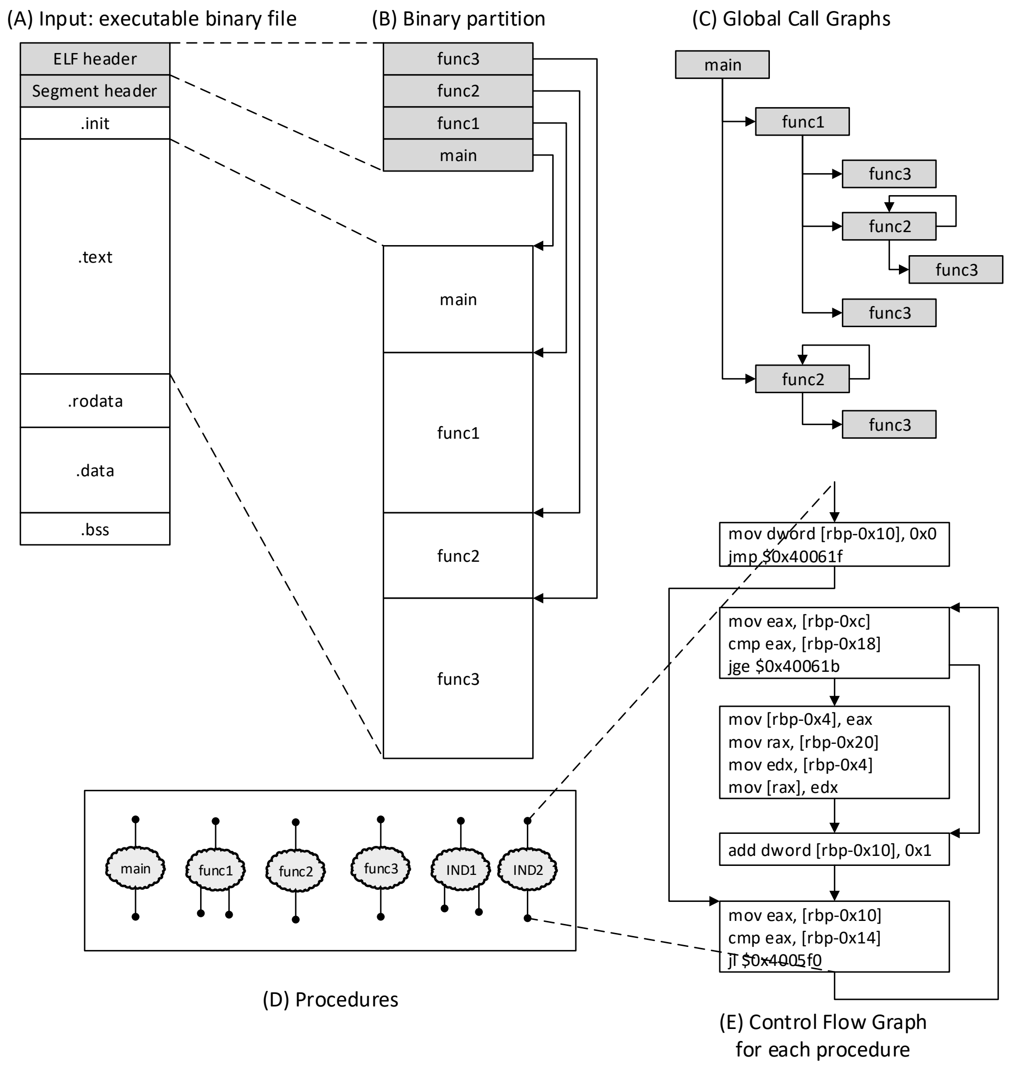


Figure 2.7: Abstraction and parsing of binary executables

Typically, an application executable is arranged in a container format, specified by an executable header at the beginning. The header is typically operating system specific (e.g. PE in Windows, ELF in Linux, Macho in MacOS) that conforms to the *Application Binary Interface* (ABI) of the operating system. The header specifies the structure of the generated machine code. A typical executable can be divided into sections such as the `.text` (executable code), `.data` (static variables), `.rodata` (static constants) and other sections according to its ABI.

The binary header may be accompanied with symbol tables that help the compiler linker to determine entry points for procedures in the executable. As shown in Figure 2.7, input executable byte streams can be parsed into regions that represent procedures according to the specification of the binary header. After parsing, the byte array for each procedure is disassembled. Basic blocks are recognised and linked based on their branches and connectivities from the disassembly. The connected basic blocks constitute a control flow graph (CFG) for the procedure. Even for most legacy stripped binaries, the symbol table might not be available. Implicit procedure boundaries can still be recovered by

x86 jump table	ARM jump table
`jmp *[0x400758+rax]`	`ldr pc, [pc+r0]`
`data case_addr1`	`const case_addr1`
`data case_addr2`	`const case_addr2`
`data case_addr3`	`const case_addr3`
`data case_addr4`	`const case_addr4`

Table 2.1: Example of instruction and data mixture: jump tables

firstly building a CFG for the whole section and then identifying isolated clusters from the CFG.

However, there are many substantial barriers that prevent an accurate binary analysis. Firstly, the mixture of instructions and data present huge challenges for disassembling the executables, especially for CISC binaries. The original compiler for the executable may encode instructions and data together for many different purposes. Some binaries are deliberately obfuscated by mixing code and data, making it difficult to analyse. Some are intended to enable more advanced architecture features such as instruction-cache alignment, branch slot optimisation, etc. For example, code generators may insert extra NOP operations to improve alignment for branch instructions. Some are used for implementing specific language constructs or avoiding the branch range limit in the architecture. A typical example is the jump table compiled from *switch statements* shown in Table 2.1, where pointers to different case statements are encoded a plain array. From binaries, determining the pattern, size and boundary for a jump table is a difficult task.

Secondly, recovering the complete CFG is difficult and error-prone due to indirect branches in binaries. An indirect jump may have multiple jump targets based on previous dynamic calculated values. If the indirect target is calculated from input data or function arguments, it is even not possible to accurately determine the target statically. Indirect jumps are very common in executables compiled from object-oriented languages such as C++. There are structures with virtual function calls encoded as indirect calls. The target of the virtual function could only be determined if the virtual function table `vftable` of the underlying class is located. There are also popular uses of function pointers and jump tables in languages such as C. The meta-information of the language constructs are typically removed as they are no longer needed for native execution. For legacy binary executables, they might also be obfuscated deliberately by their original venders for security purposes or space limitations. The obfuscation typically increases the uses of indirect branches. For example, Figure 2.8 shows a normal function `call` is obfuscated into indirect calls on the stack.

Thirdly, analysing the data flow on top of the problematic CFG is more tedious and error-prone. Modern architectures such as x86 typically have hundreds of different instructions, with new extensions added at each processor revision. Each instruction may have complex semantics, making it difficult to obtain data flow with millions of these instructions. Some hardware features may result in more ambiguity in generated binaries, such as indirect memory accesses, conditional instructions, software exceptions, etc. Indirect memory accesses present a huge challenge for analysing data dependencies from memory accesses in executables. There are existing analysis algorithms such as Alias Analysis [40] or Value Set Analysis [41] that perform over-approximation to the set of

Binary without Obfuscation

```
call function
```

Binary with Obfuscation

```
push function_addr
sub rsp, 0x20
...
jmp *[rsp+0x20]
```

Figure 2.8: Example: the call to `function` is obfuscated through stack operations

values that the instruction may access.

All the mentioned factors constitute in the loss of information compared to the information that can be retrieved from source level. Therefore, the scope and accuracy of resulting static analysis would be affected. Granted, heuristics and architectural conventions can partially solve some of these problems. Static binary analysis fails to achieve moderate accuracy for generic executables.

2.2.2 Dependence Analysis

In order to transform the original binary with the same expected output, the static analysis is primarily a problem of studying *dependencies* between instructions. A dependency refers to a set of order constraints to correctly execute an instruction in the program. The enforcement of all dependencies guarantees the observation of the same program behaviour. It is possible to derive many different execution orders by respecting the same dependencies in a program. The orders are proved to be strongly equivalent to each other [42].

The specification of dependencies is arranged in data structures such as *dependence graphs*. Consecutive dependencies on the same storage entity (memory addresses etc.) make up the *flow* of information. Typically, the flow analysis of a program consists of studies in *control flow* and *data flow*. The results of control and data flow analysis are used for constructing the whole program dependence graph (PDG).

Control Dependencies

Ideally, the control flow analysis of the executable is performed on the control flow graph (CFG) of the executable, whose nodes are individual instructions. For simplicity, sequences of instructions ended with only one control transfer instruction are wrapped in a *basic block*. All instructions in a block are assumed to be executed in order in an in-order machine. With the simple control nature in basic blocks, control flow graphs (CFG) can be simplified, where each node in the CFG represents a basic block, and each edge in the CFG stands for a control transfer.

A complete CFG for the whole executable is typically hierarchical. It is arranged into two levels of directed graphs as shown in Figure 2.7. The first level is called *Call Graph*, which represents calling relationships between subroutines in the executable. Each node accounts for a subroutine, and each edge (`f`, `g`) indicates that procedure `f` calls procedure `g`. Thus, cycles in the directed call graph represent recursive procedure call chains. The second level denotes the CFG for each subroutine whose nodes are basic blocks. The whole CFG for each procedure represents a node in the first-level call graph.

A node `N` is said to have a *control dependence* on a preceding node `M` if the outcome of `M` determines whether `N` should be executed. However, edges in a CFG do not explicitly reflect the control dependent relations from any two blocks from the CFG. Cytron et al. [43] gave a formal definition of control dependencies using dominator analysis.

For a CFG with an entry node E, suppose there are two nodes `M`, `N` from the CFG, we say node `M` *dominates* node `N` when all paths from node E to node `N` must also go through node A. The *immediate dominator* of `N` refers that there is only one distinct node `M` that dominates `N` and there is no other node in the graph that also dominates `N`. By combining domination relations for all nodes in a CFG, a *dominator tree* could be built, where children of a node are those nodes it immediately dominates. The dominator tree can be used for quickly querying the dominating relations between any two blocks from a CFG.

The *dominance frontier* of a node `M` refers to a set of nodes `N`, where `N` is not dominated by `M`, but `N`'s predecessor is dominated by `M`. It is a set of nodes that `M`'s dominance terminates. The dominance frontier is typically used in compilers to convert statements into static single assignment form. For some cases, a *post-dominance* relation is defined as the reverse of dominance analysis, where it traverses from one of the exit nodes to the entry node in the CFG. From Cytron et al., the formal definition of control dependencies can be defined as: Let X and Y be two nodes in a control-flow graph. Y is control dependent on X `iff` X is in the *post-dominance* frontier of Y. The control dependence relation is not symmetric, reflexive, nor transitive.

Data Dependencies

A program consists of many instructions where each instruction processes multiple sources into one or multiple destinations. The flow of data is created when multiple instructions reuse data from the same storage location. A data dependence defines the direction of the flow between storages.

For two instructions i and j in the program that access the same a storage location M, different types of data dependence are created from j to i:

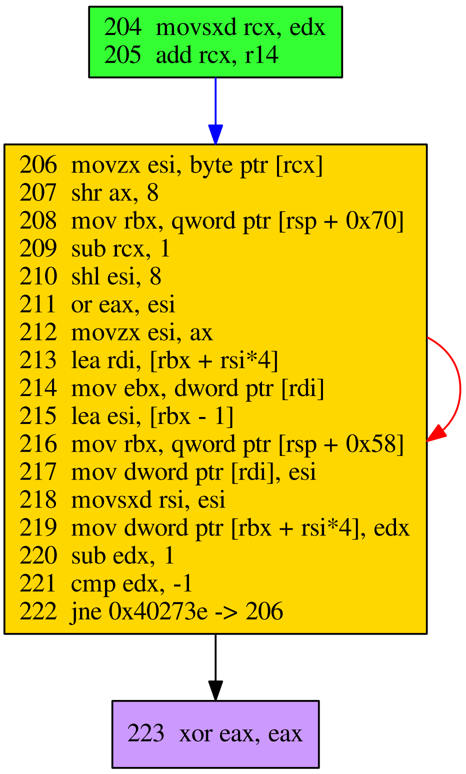
1. Read-After-Write(RAW), if i writes to M and j reads from M. It is called the flow or true dependence since the data flows from i to j through the storage location M.

2. Write-After-Read(WAR), if i reads from M and j writes to M. It is called anti or name dependence. This kind of dependency is normally caused by reuse of storage locations, and they can be removed by renaming (privatisation) or using other storage locations.

3. Write-After-Write(WAW), output dependence: if both i and j write to M. It is also called the output dependence and they can be also removed by renaming.

Figure 2.9 shows an example of the different kinds of data dependencies from static binary analysis. At binary level, A data flow *edge* is formed when two instructions access the same register, stack or memory locations. The combination of all data dependence edges in a procedure constitute a data dependence graph (DDG), where each edge is one of the RAW, WAR and WAW types.

Retrieving accurate DDGs from binaries is a challenging task. It has been a body of intense research for decades [44] in compiler optimisations. The conventional approach

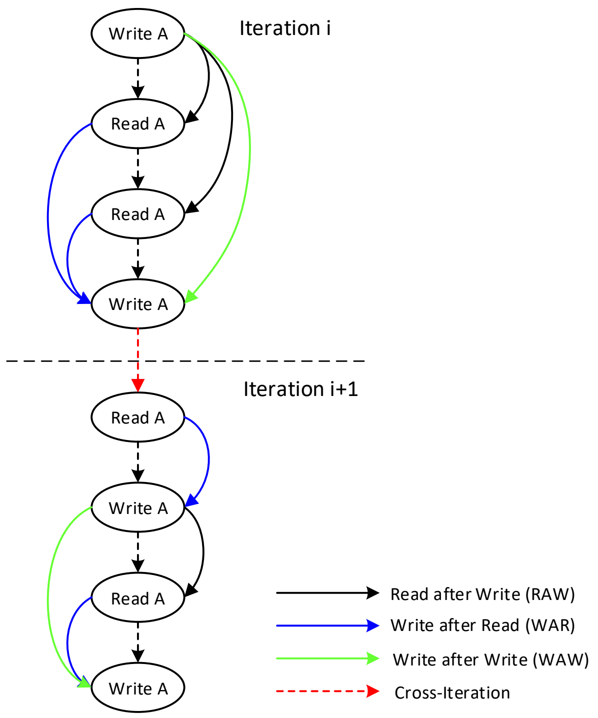


```
204  movsxd rcx, edx
205  add rcx, r14
```

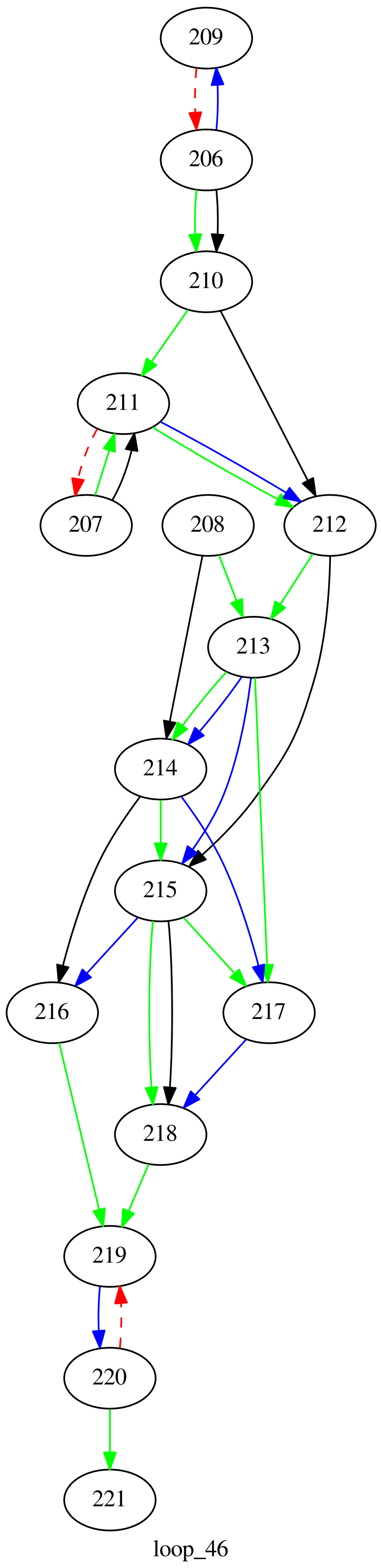
```
206  movzx esi, byte ptr [rcx]
207  shr ax, 8
208  mov rbx, qword ptr [rsp + 0x70]
209  sub rcx, 1
210  shl esi, 8
211  or eax, esi
212  movzx esi, ax
213  lea rdi, [rbx + rsi*4]
214  mov ebx, dword ptr [rdi]
215  lea esi, [rbx - 1]
216  mov rbx, qword ptr [rsp + 0x58]
217  mov dword ptr [rdi], esi
218  movsxd rsi, esi
219  mov dword ptr [rbx + rsi*4], edx
220  sub edx, 1
221  cmp edx, -1
222  jne 0x40273e -> 206
```

```
223  xor eax, eax
```

Loop 46 in 401.bzip2:mainSort



(a) Illustration of four kinds of data dependencies



(b) Data Dependence Graph

Figure 2.9: Example of a small loop from an executable from the SPEC2006 401.bzip2 benchmark

to perform data-flow analysis is to set up dataflow equations for each node of the CFG and solve them by repeatedly calculating the output from the input locally at each node until it converges. The dataflow equation of a node specifies a *transfer function* and *join function* between the input and outputs of its predecessors in the CFG.

$$\texttt{OUT}_\texttt{n} = \texttt{Transfer}(\texttt{IN}_\texttt{n}) \tag{2.1}$$

$$\texttt{IN}_\texttt{n} = \texttt{Join}(\cup_{p \in \texttt{pred}} \texttt{OUT}_p) \tag{2.2}$$

However, due to the widespread existence of memory indirection (pointers), it is impossible to statically determine the exact read and write locations for some instructions[45]. For instructions with indirect accesses, transfer and join functions are inherently ambiguous. Hence they affect the accuracy of data flow analysis. Alias analysis is the process to prove whether two indirect (pointer) accesses that are reading the same location. For the last three decades, many algorithms and approaches [46, 47, 48, 49, 50, 51] have been developed to boost the accuracy of alias analysis.

However, many comprehensive alias analysis suffers diminishing returns if more sophisticated analysis algorihm is used. The existence of pointer arithmetic and casting makes it an NP-hard problem [52]. Moreover, there are many parts of the code that is directly referencing to input data, which are statically undecidable on dependencies. Even with decidable data dependencies, for programs with dynamically changing access patterns, the static analysis could not conclude a close-form dependence relation. Consequently, the result of static data dependence analysis is typically a conservative approximation. The conservative analysis would potentially turn off many transformation and optimisation opportunities.

Loop Dependencies

Specifically for loops, dependencies can be further categorised. Dependencies generated by instructions from different iterations in a loop are called *intra-iteration* or *cross-iteration* dependencies. Dependencies occur between instructions in the same iteration are called *inter-iteration* dependencies. For example, Figure 2.10 shows the updated dependence graph by unrolling the loop from Figure 2.9 into dynamic instances. Loop related dependencies are essential to enable fundamental transformations in automatic parallelisation on loops. More details in recognising loop dependencies will be discussed in the next section.

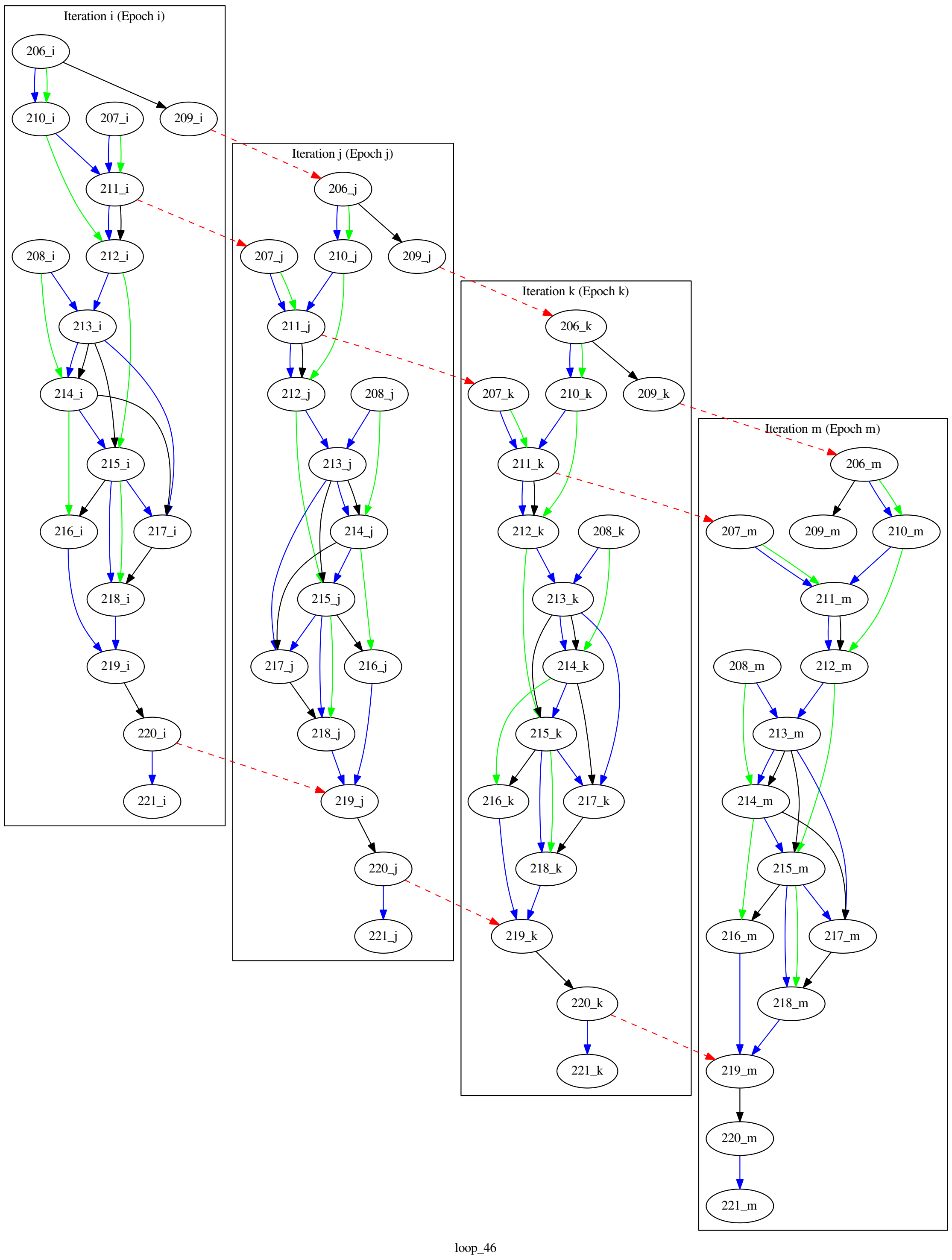


Figure 2.10: Dependence Graph of dynamic instances of instructions by unrolling the loop from Figure 2.9 into dynamic instances

2.3 Automatic Parallelisation

The third field of related work is *automatic parallelisation*. Parallelisation refers to converting sequential, single-threaded executions into the multi-threaded executions on a parallel platform. The process of parallelisation can be manual, interactive or fully automatic. Despite that manual or interactive parallelisation of a program is challenging and time-consuming, automatic parallelisation is the most ambitious task of the three. Its objective is to remove the burden of programmers on understanding and expressing the parallelism which exists in the algorithm.

Automatic parallelisation research for parallel hardware has a long history since the introduction of parallel programming. Conventional automatic parallelisation techniques rely on results of static program analysis to recognise parallelism, and transform the identified sequential code into a parallel form. The correctness is realised by enforcing data and control dependencies collected from static analysis or profiling information. As applications spent most of the executing time in loops, most techniques focus on loop-level parallelism.

Despite the enormous effort in research, a generally applicable solution for automatic parallelisation is still elusive. The most difficulty resides in the automation of the process without any manual intervention. Compared to manual parallelisation, current generic automatic approaches lack the ability to parallelise a program based on its algorithm or data structures. They have to follow fixed loop models based on the nature of data and control flow of the loop. Based on the way of decomposing parallelism, three primary loop parallelisation models have been developed: DOALL[53], DOACROSS[54] and DOPIPE[55]. These models are also called *independent*, *cyclic* and *pipelined* multi-threading respectively.

2.3.1 Independent Multi-threading

A loop is called DOALL, *independent* or *embarrassingly parallel* if each iteration in the loop can be executed independently without interfering with other iterations. DOALL loop iterations can be concurrently scheduled to threads with no order requirement. No synchronisation is required, and consequentially it yields a proportional speedup.

DOALL loops are the simplest loop form and they are typically observed from scientific, media and image based applications. Even for manual parallelisation, competent programmers tend only to parallelise this form of loop since it is easy to implement without writing synchronisation code and the performance gain is almost proportional. Due to its simplicity, there are also a few automatic parallelisation compilers such as Polaris [56], SUIF [11] that focus on DOALL and DOALL-like loops.

From the static analysis, a DOALL loop can be recognised if we can either statically or dynamically prove there are no other cross-iteration dependencies except the induction variable. However, it is very difficult to prove the independencies across iterations, for a complicated loop with indirect memory accesses, function calls and other sophisticated control flows. With limited power of static analysis, the coverage of statically-proved DOALL loops is typically low. There is a range of dynamic approaches such as the LRPD test [57] that parallelise undecided loops to be DOALL loops speculatively and optimistically. More details will be discussed in Section 2.3.5.1.

To extend the scope of DOALL loops, a broader concept for DOALL is used by many automatic parallelisation compilers. Figure 2.11 illustrates a typical DOALL loop

DOALL loops

```
for (i=0; i<n; i++)
{
    a[i] = work(i);
}
```

DOALL loops with reduction variables

```
int result, val;
int sum = 0;
for (i=0; i<n; i++)
{
    result = work(i);
    val = evaluate(result);
    sum += val;
}
```

Figure 2.11: Example of DOALL loops if the function `work` and `evaluate` are proved independent.

and a DOALL loop with clear induction and reduction variable with simple reduction operations. The rest of accesses are independent between each iteration. It allows the removal of WAR and WAW cross-iteration dependencies using *privatisation* techniques [58]. Reduction variables on a few simple reduction operations (plus, subtract, etc.) can also be handled using thread-private reductions.

Operations allowed in the broad-DOALL loops can be summarised as a function with the iteration index as its argument. In other words, a thread can regenerate all relevant iteration context based on the given value of induction variable instead of relying on the previous versions of variables. The transformation is the reverse of *strength reduction* in compiler optimisations. With this property, the iteration space of DOALL loops can be split and scheduled freely to different execution units without synchronisation.

2.3.2 Cyclic Multi-threading

Even with the broader definition of DOALL loops, the scope of DOALL loops is still limited. With the existence of many cross-iteration dependencies, a more generic loop model for automatic parallelisation called DOACROSS [54] or cyclic multi-threading was developed. The DOACROSS model assigns loop iterations to fixed number of threads in a round-robin style. All cross-iteration dependencies are enforced by forwarding values between threads through synchronisation. For each cross-iteration dependence pair, each thread waits and retrieves the input from the previous thread, performs calculations, and signal the updated value to the next thread. The minimum required number of instructions to complete the wait, calculate and signal is treated as a *sequential segment*. Parallelism can be achieved by overlapping the execution of the rest of independent code.

For example, Figure 2.12 shows the flow of DOACROSS operations. Cross-iteration dependencies between from block D to block C is resolved by forwarding output data through thread synchronisation. Performance can be achieved by overlapping the execution of the rest of loop body (block E, A, B) in parallel. The proportion of the size of the sequential segment over the whole loop iteration size reflects the sequential fraction of the loop.

DOACROSS is universally applicable and scalable. Any generic loop can be transformed into the form of one or more sequential segments followed by parallel segments, as long as the data dependence graph is obtained. However, DOACROSS loop models

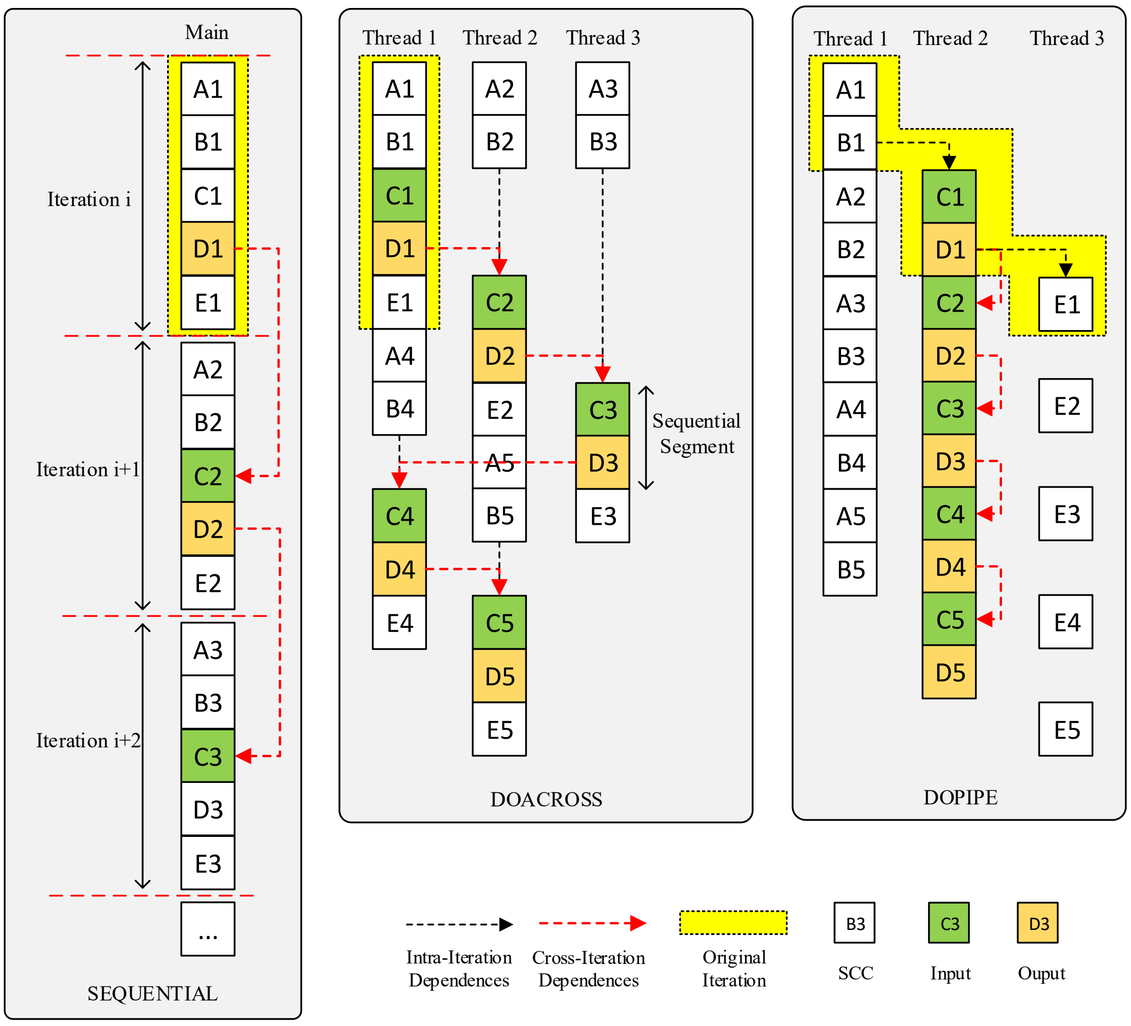


Figure 2.12: DOACROSS and DOPIPE parallelisation

have a few disadvantages. Firstly, if the size of sequential segments is too large compared to the whole iteration size, the loop is not beneficial for parallelisation since most of the operations have to be serialised. Secondly, DOACROSS is highly sensitive to inter-thread communications. For each iteration, it has to execute one or more sequential segments that require at least two synchronisation operations. The critical data flow path is placed through synchronisation, so that a single fluctuation in thread communication would affect the execution of all subsequent waiting threads. High communication cost would easily amortise the benefits provided from overlapping of parallel fractions. In the past when inter-thread communication was expensive (hundreds of cycles), DOACROSS could not find suitable loops that demonstrate good performance.

In recent years, the emergence of the single-chip multicore processors has drastically reduced the communication costs, inspiring HELIX [59] to extend the DOACROSS model to achieve performance for more general loops. Compared to DOACROSS, HELIX minimises the size and number of sequential segments using comprehensive static analysis. It also reorders sequential segments to achieve better overlapping. To further reduce the cost of thread synchronisation, it uses hyperthreading to pre-fetch thread communication signals. HELIX demonstrated an average 2.25× speedup for SPEC2000 benchmarks [60] on a six-core commercial processor, which is one of the most efficient automatic parallelisation techniques appeared to date in the area. As the cost of thread synchronisation continues to shrink due to advances in parallel hardware and process technology, DOACROSS style parallelisation will achieve performance on more loops in practice.

2.3.3 Pipelined Multi-threading

Another universal parallelisation paradigm is the DOPIPE or pipelined multi-threading. Rather than placing each iteration cyclically on each core, DOPIPE splits the loop body into several stages. Each thread is only responsible for executing dynamic instances from a single stage. Figure 2.12 (right) shows the loop iteration is split into three stages, AB,CD and E. And all three stages are scheduled to three different threads respectively. Compared to DOACROSS, DOPIPE arranges the original thread communication path $D \rightarrow C$ to the same thread and exposes the original inter-iteration dependencies to communicate across threads.

Decoupled software pipelining (DSWP) [10] is a state-of-the-art DOPIPE technique. To apply the orthogonal transformation, DSWP performs extensive static analysis and builds a dependence graph for the static form of loop iterations. It then splits the graph into stages and schedules them into multiple execution units. Since the slowest stage determines the overall performance, it is essential to ensure load-balancing between threads for optimal performance. DSWP demonstrated an average of 1.81 $\times$ speedup on eight cores for SPEC2000 benchmarks. There are also a few extensions [61, 62] that combine other forms of parallelism into their paradigm.

The principle of DOPIPE is to reduce the inter-iteration (cross) dependencies by applying an orthogonal transformation between the original intra-iteration and inter-iteration dependencies. If the resulting transformed code turns out to have more inter-iteration dependencies across thread contexts, it is then not beneficial to apply the pipelining transformation. For many complicated loops from generic applications, DWSP technique finds it difficult to split effective partitions and achieve a balanced load for threads. Moreover, the scalability of the stage partition is also limited. There are typically not many stages that can be effectively partitioned. It is typically difficult for DSWP to scale beyond four threads [61].

2.3.4 Polyhedral Multi-threading

Both DOACROSS and DOPIPE are universally applicable to all kinds of loops. For DOACROSS, it splits workloads by following the boundaries of the original iterations, while DOPIPE performs an orthogonal transformation and partitions tasks into stages. They both suffer the excessive synchronisation cost if there are more dependencies that cross the boundary of partitions. Between the two approaches, DOACROSS is more sensitive to the number of cross-boundary dependencies while DOPIPE is more vulnerable to load balancing.

It is possible to perform a transformation in a specific direction in-between the cross-iteration and intra-iteration space, so that the transformed code would have the minimum number of synchronisations across the boundary of partitions. However, transformation to another iteration space is not easy for general loops if the access patterns of a loop cannot be summarised in a close form. Instead, people focused on a very narrow spectrum of loops called *affine loops*. Affine loops are multi-dimensional loops that have uniform loop

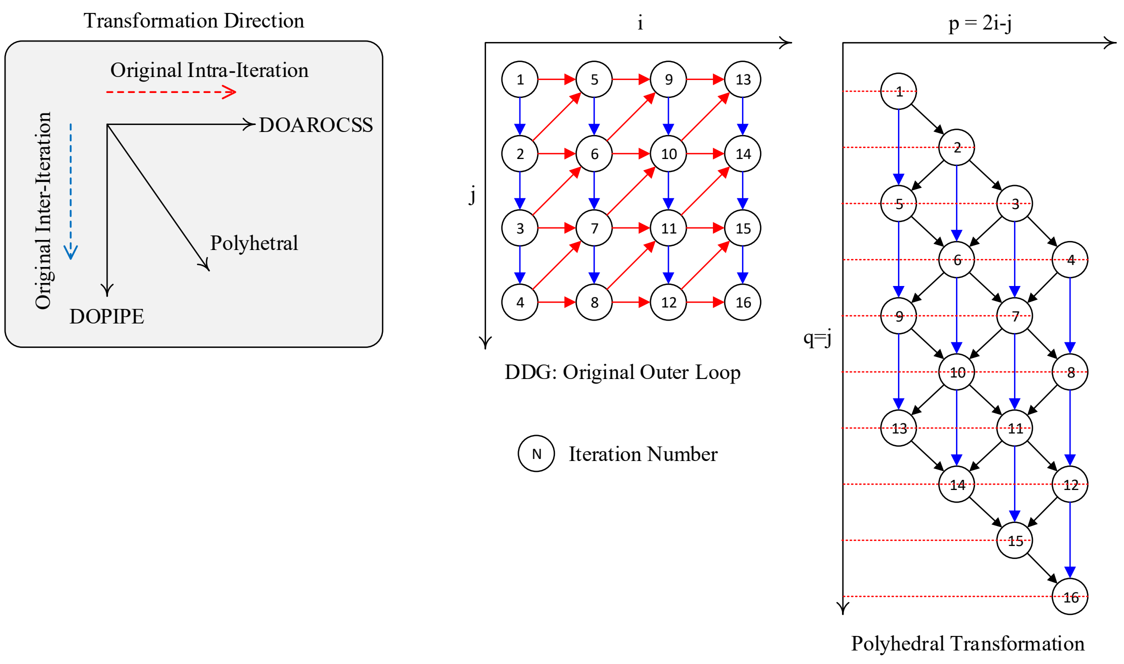


Figure 2.13: Parallelism exposed by polyhedral transformation

induction variables and can only have memory accesses whose address must be indexed by a polynomial representation of loop induction variables. No complex control flow is allowed in the loop. Affine loops are commonly seen from signal or image processing applications that operate on matrix data structures. The following code shows an example of an affine loop.

```
for (int i=0; i<M; i++) {
    for (int j=0; j<N; j++) {
        A[c1*i+c2][c3*j+c4] = B[c5*i+c6][c7*j+c8];
    }
}
```

With uniform loop induction variables and polynomial memory accesses, data dependence relation could be analysed precisely in the space of iteration dimension (i, j). The data dependence mesh is characterised as *dependence polyhedron*. The dependence polyhedron on coordinates (i, j) can be *affine transformed* into another space (p, q) so that dependencies in the new coordinate system are minimised.

The polyhedral scheduling approach provides a framework that unifies DOALL, DOPIPE and DOACROSS. PLUTO [12] and LLVM Polly [13] are compilers that enable polyhedral parallelisation. Despite the effort to extend the model to be more applicable [63], polyhedral models could hardly find any optimisation on generic applications with irregular loops. As a result, polyhedral parallelism is not considered in this dissertation.

2.3.5 Speculative Multi-threading

All the DOALL, DOACROSS and DOPIPE loop models are based on the assumption that static analysis would provide accurate information of data dependencies. However, as discussed in Section 2.2.2, it is difficult to obtain accurate data dependence relations, especially from loops with many complex control flows and indirect memory accesses. Historically automatic parallelisation techniques [56, 11] reject ambiguous loops for parallelisation. HELIX[59] and DSWP [10] shrink the dependence ambiguity by employing more aggressive and expensive data dependence and alias analysis [49]. It is also shown that with improved accuracy of static analysis, the consequent performance is improved [10]. However, from the level of HELIX static analysis, it has seen diminishing returns as more compute intensive static analysis is employed [64].

2.3.5.1 Thread Level Speculation

In contrast to seeking the extreme of accurate static analysis, an alternative approach is to use *thread-level speculation* (TLS) or *speculative multithreading*. The concept of speculation is well adopted in the architecture design for instruction level parallelism, such as *speculative execution* in superscalar architectures. Similarly, TLS is a run-time technique to allow tasks to be executed concurrently by optimistically ignoring undecided dependencies. If there turns out to be a conflict caused by a dynamic data dependence, all changes made by the thread are aborted and the control of the execution reverted to restart the speculative code section.

TLS allows the parallelisation of a program without prior knowledge of data dependencies and avoids the need for absolutely accurate information for data dependencies. It removes the burden of heavy static analysis but at a high cost of mis-speculation. Therefore, TLS is only beneficial when undecidable dependencies are independent or rarely conflict during the execution. If the undecidable code region is frequently in conflict, the mis-speculation cost would be too high to deliver performance.

To support speculation at thread level, the runtime system must include two essential features: *version management* and *conflict detection*. Version management is the ability to isolate the contexts of a thread execution, so that all writes of an aborting thread can be squashed or undone. By version management, there are two policies: *lazy* or *eager* versioning. For lazy versioning, each speculative thread buffers all its writes until its commit, which is an effective way to provide privatisation to remove WAR and WAW dependencies and facilitate reverting changes without polluting shared data. For eager versioning, each speculative thread can directly modify the memory but needs to maintain an additional undo log.

The second feature is the runtime conflict detection. Depending on the implementation, there are also *lazy* and *eager* detections. Lazy conflict checking buffers and delays all accesses and validates them as a whole during the commit operation. While the eager conflict detection performs checking at each speculative access. The difference of the two conflict detections is shown in Figure 2.14. Lazy conflict checking saves the cost of performing checks at every access, but in turn, it results in a larger context squash and re-execution cost in case of mis-speculation.

The support for TLS can be implemented in either hardware and software. For hardware support, the pioneering research was to reinvent a completely new CPU architecture to support speculative task parallelism, such as the MultiScalar Processor[65], the Standford Hydra Processor[66], the Trace Processor [67] and the CMU STAMPede Processor [68]. These TLS architecture could effectively achieve considerate performance from speculative threads over a series of research benchmarks. However, the performance gain is not enough to justify the huge cost of manufacturing an entirely new CPU architecture.

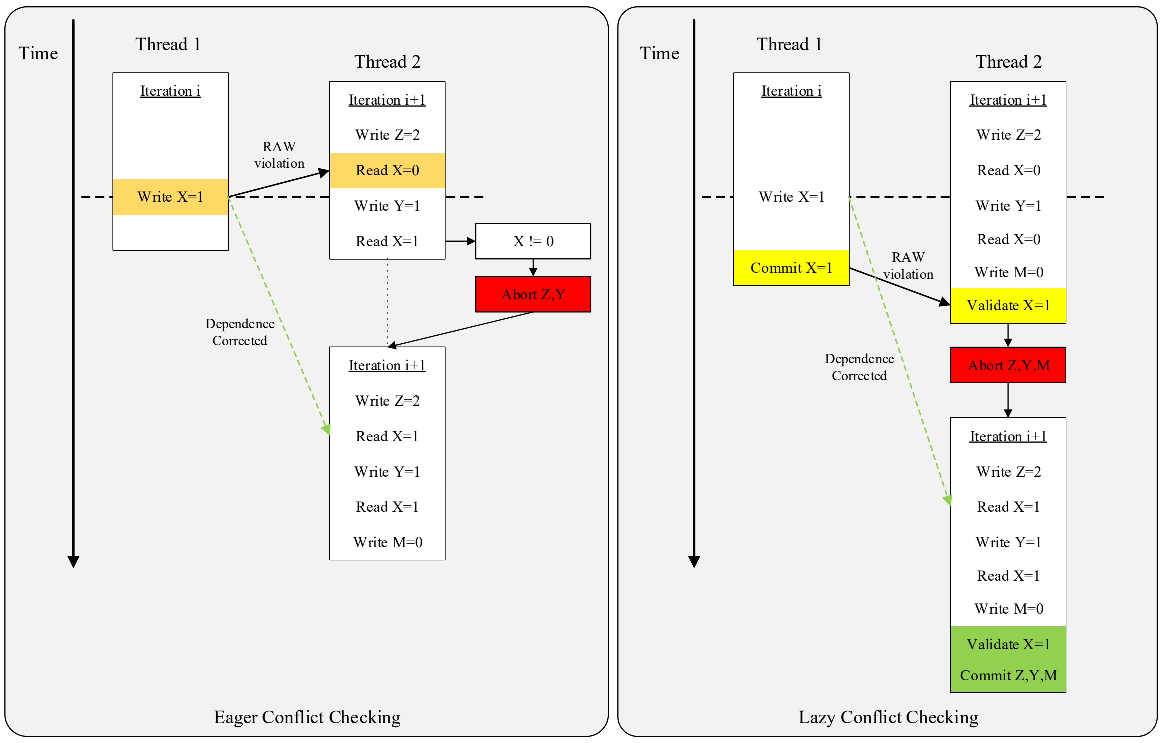


Figure 2.14: Two different conflict detection schemes in TLS: eager and lazy conflict checking

Therefore there is not a TLS processor that is commercially implemented. The new hardware also requires significant changes in compilers to support TLS. TLS compilers such as POSH [69] and Mitosis [70] not only need to recognise profitable tasks that can be scheduled speculatively but also have to handle corner cases such as exceptions, system IO etc.

On the other hand, software-based TLS offers more flexibility and runs without additional hardware extensions. The first well-known software-based TLS is the LRPD Test [57]. The work firstly selects and transforms FORTRAN loops with privatisation and reduction optimisation. Then the loop is speculatively executed as DOALL loops while all ambiguous accesses are recorded. After the loop has been completed, a dependence test is applied over the recorded access to ensure the loop has no cross-iteration dependencies. If the test fails, the loop is sequentially re-executed, otherwise, the parallelisation is considered successful. The flexibility of software-based TLS results in a high price of high overhead in check-point machine contexts and re-execution if mis-speculated. The bookkeeping for conflict checking also requires additional computing resources, which implies performance degradation.

2.3.5.2 Transactional Memory

While the idea of thread-level speculation was originated from computer architecture researches to exploit thread-level parallelism from both hardware and software, *transactional memory* (TM) [71] emerged from the field of concurrent programming, which was to address the problem of lock-based parallel programming. Conventional parallel programming using locks is faced with many problems such as performance, contention and verification difficulties. Therefore, it is typically difficult to write a correct and efficient parallel program using lock-based programming. By using transactional memory, programmers can surround critical regions in a *transaction* without specifying software locks. By carefully selecting regions for speculative execution, data dependencies in the transaction can be resolved implicitly and automatically. Without worrying about the problems caused by lock-based programming, programmers can continue to write parallel code in a sequential style while maintaining consistency. Therefore TM has inspired many lock elision researchers [72, 73] to migrate from conventional lock-based parallel programming to a much simplified transaction-based programming.

Both TLS and TM converged to follow the same speculation principle, and thus implies a similar implementation. However, the difference between a TM and a TLS system is that TM does not require specific ordering for commit operations. Therefore, the TM system has an additional *contention manager* to decide which thread needs to abort in case of conflicts. Secondly, TM is more like a programming paradigm which involves programmers to manually and explicitly spawn threads and select regions for parallelisation. While TLS is mostly an automatic approach that spawns thread under the hood without notifying the users. Thirdly, the granularity of TM could be as large as objects, which offers more flexibility over conflict checking in user specified data structures. On the other hand, TLS is only limited to contention detection at the granularity of mostly word and cache-line levels.

Hardware transactional memory

Similarly to TLS, transactional memory can be either implemented in hardware (HTM) or software(STM). For hardware transactional memory, speculation could be supported implicitly or explicitly. On the one hand, for implicit HTM, it requires programmer or compiler to specify the boundary of a speculative transaction, two extra instruction `begin_transaction` and `end_transaction` are added. All instructions between the two special instructions are executed implicitly and speculatively. On the other hand, explicit HTM introduces new instructions for speculative load and store, which supports free interleaving between speculative and non-speculative accesses. With this difference, implicit HTM can support executing arbitrary code in the transaction, including legacy library code, while explicit HTM has to recompile to support new speculative load/store instructions. However, due to the explicit specification of accesses, explicit HTM provides programmers with more flexibility, and it also results in a smaller transactional size.

Compared to fundamental architectural change for hardware TLS, HTMs are typically implemented as an extension to existing architecture. Data cache becomes a natural place to detect dependence conflicts since all memory accesses involve cache lookups, therefore a few HTMs [74, 72, 75] are developed on caches by adding new speculative bits to the cache line and modifying the cache coherence protocol. There is other work [76] that uses dedicated hardware buffers to serve as read and write sets. Currently there already exist commercial processors that implements HTM, such as the Intel TSX [77, 78], Sun Rock HTM [75] and IBM Blue Gene Q [79]. However, HTMs are typically limited in transaction size due to hardware capacity. Despite the fact that there are hybrid HTM [80, 81] that offer unbounded transaction size, performance offered by HTM was not high enough for broad application in parallel software development.

Software transactional memory

In contrast, software transactional memories (STM) support unbounded, nested and conditional transactions, offering flexibility and compatibility on existing hardware platforms.

Instead of requiring additional hardware support, STM maintains its meta-data or logging to perform thread isolation, conflict detection and management. Since there is no hardware support that provides quick lookups, a hash function is typically used to map arbitrary speculative access to the corresponding meta data. Typically the search would incur extra computational overhead for each access, implying a performance penalty if the transaction size is large enough.

Similarly to the implementation of TLS, there are different design choices for STM features. As for version management, STMs can be either *lock-based* or *value-based.* For conflict detection, it can be either *eager* or *lazy*, which implies its speculative writes to be either *directly* updated or *deferred* updated on the shared data.

The most famous lock-based TM algorithm is the *transactional locking II* (TL2) [82], which uses an array of locks to represent the whole address space. Each lock has a version number indexed by a shared global version clock. Upon a speculative read, it finds the corresponding lock from a lookup and simply records the version of the lock for the location. For speculative writes, TL2 defers all its writes in a thread-local write buffer. When a transaction is pending commit, TL2 acquires locks for all its writes and validates that all lock versions from all its writes and reads are unchanged. If any of the lock version turns out to be outdated, it then aborts the transaction. After validation, it safely releases each lock with a higher version of each lock. There are also other lock-based STM such as McRT-STM [83] and Bartok-STM[84] that uses direct updates so that each speculative write needs to acquire the corresponding lock, write the value, increment the version of the lock and back up the original value into an undo log.

There are also other STMs which are focusing on providing lock-free and obstruction-free concurrent programming models. These STM typically operate on coarse-grained objects and incur higher operation overhead in acquiring locks and validations. However, few of them can be used to support thread-level speculation. JudoSTM [85] is one of few *value-based* STMs optimised for TLS. Within a transaction, each thread copies its initial read into its read set and buffers all its writes into its write set. It then validates all read values before committing its write buffer to shared memory. Any read violation would cause it to drop its write set and rollback to the start of the transaction for re-execution. The principle of value-based STM is simple, and the code could be easily generated and inlined into the original application instructions. In this dissertation, I present a new implementation of STM dedicated to minimise operation cost using Just-In-Time compilation. The design choices and implementation will be discussed in Chapter 5.

2.3.6 Profile Guided Multi-threading

With the help of thread level speculation, automatic parallelisation techniques could overcome ambiguous and undecided data flow obstacles in the static analysis. However, parallelisation using TLS is only effective and beneficial when the rate of mis-speculation is low. Frequent violations in data dependence would cause frequent aborts and re-executions, which is a waste of computing resources and results in a huge performance penalty. Instead of performing speculation on the frequent violated locations (FVL), it would be less expensive to forward the value through synchronisation. For undecided data dependence pairs, it is statically impossible to determine the occurrence probability of data dependence violation. Therefore it is difficult to select beneficial regions of code that manifest low mis-speculation rate from static analysis.

Profiling technique is an emerging technique to assist static analysis to overcome the problem. It collects runtime information about the data dependence graph of a program from its past execution with training inputs. Information about the crossing-iteration data dependence probabilities could effectively help the decision making in selecting loop regions for thread level speculation. There are a few work [86, 87, 88] that built different cost models from profiling information to select the most profitable loops for TLS. Alias profiling [86, 89] has been proposed as an assist for memory disambiguation for data dependencies in the static analysis. The cost model for task recommendation is based on the self length that models task sizes, dependence length that models conflicts, and a speedup estimate. Most of the profiling information is collected from execution traces. There are also profiling tools [90] that are based on binary instrumentation.

However, the drawback of profiling information is the input-dependent problem. Profiling information for one run is associated with the given input. The information might not be applicable for the same program with other inputs. Therefore the profiling information is not reliable to use in a general parallelisation model. In chapter 5, we also investigate how we can augment this drawback with the benefits of thread level speculation.

2.4 Summary

This chapter discusses three essential fields of related work to build a binary recompilation and parallelisation infrastructure: dynamic binary translation (DBT), static binary analysis (SBA) and automatic parallelisation (AP) techniques.

Firstly, DBTs offer flexibility and easy access to runtime information compared to static binary translation. Code-cache based DBTs provide Just-In-Time (JIT) recompilation, dynamic code optimisation and light-weight instrumentation capabilities. It is, therefore, reasonable to implement a binary recompilation tool based on the code-cache-based DBTs. However, the recompilation is typically restricted and short-sighted, due to the lack of a global view of applications.

Secondly, SBA reasons about global structures of applications without running them. However, many substantial barriers prevent an accurate binary analysis, such as indirect branches, code obfuscation, the mixture of code and data, etc. Control and data dependencies are essential to maintaining correctness for a proposed transformation for optimisation. However, due to the existence of memory indirection, dependence information is also difficult to retrieve.

The combination of SBA and DBT augments each other's strength and weaknesses. SBA can resolve the lack of global understanding of programs in DBT. In turn, DBTs have runtime information to address the ambiguity problem in SBA. In chapter 3, I will discuss how SBA and DBT can cooperate to achieve complicated binary recompilation.

Thirdly, conventional automatic parallelisation techniques have three loop models: DOALL, DOACROSS and DOPIPE. They all rely on accurate dependence information for correct decomposition and transformation of the original sequential code. The inaccuracy of static analysis limits the applicability and scalability of the models. Thread-level speculation can address the problem of ambiguity, however, it suffers many implementation constraints in both hardware and software. Profiling information can also alleviate the uncertainty in the static analysis. However, it is not reliable since it is input-dependent.

Chapter 3

Recompiling Binaries As Instructed

As more *Just-In-Time* capabilities, such as full encoding components, have been added in dynamic binary translation (DBT) tools, the research focus on binary translation has shifted to binary modification and instrumentation. With JIT capability, users are free to modify or insert instructions as the application is running. The scale of user-level dynamic binary modification evolved from small callback changes to comprehensive cross-block modifications. DBTs based on code caches typically buffer their modifications. It reminisces the same principle as JIT compilers where native code is lazily compiled and buffered only when it needs to be executed.

However, JIT compilers typically rely on a form of intermediate representation (LLVM IR) or byte code (JVM, .NET and DVM) to perform compilation in a very short time. Due to the lack of cross-block symbolic information, modifications in DBTs are typically limited at the granularities of basic blocks in a linear control flow fashion. As discussed in chapter 2, the combination of static binary analysis (SBA) and DBT augment each other's strength and weaknesses. SBA can resolve the lack of global understanding of programs in DBT. In turn, DBTs have runtime information to address ambiguities in SBA. Inspired by this idea, we can use static binary analysis and encode the information into a structured hint format. The hint information can be efficiently interpreted by the DBT to perform large-scale consistent modifications.

This chapter presents a novel binary recompilation framework called *Guided Binary Recompilation* (GBR) that uses static binary analysis to guide dynamic binary modifications on binaries. GBR does not require the availability of original source code or extra debugging information. While this dissertation focuses on configuring GBR for performance improvements on single-threaded binaries, GBR is a scalable and accessible platform for many other purposes, such as profiling, debugging and instrumentation.

3.1 System Overview

To show how an executable is automatically recompiled in GBR, consider Figure 3.1. During dynamic binary translation, for each incoming new basic block, there is a modification oracle that instructs the translator to perform exact modification operations. If the guidance is fully correct and the binary translator strictly modifies the block as instructed, the combination of all per-block modifications constitute a global recompilation across the binary. With this assumption, any complex binary modification can be decomposed into a set of small and simple modification passes. The modification passes

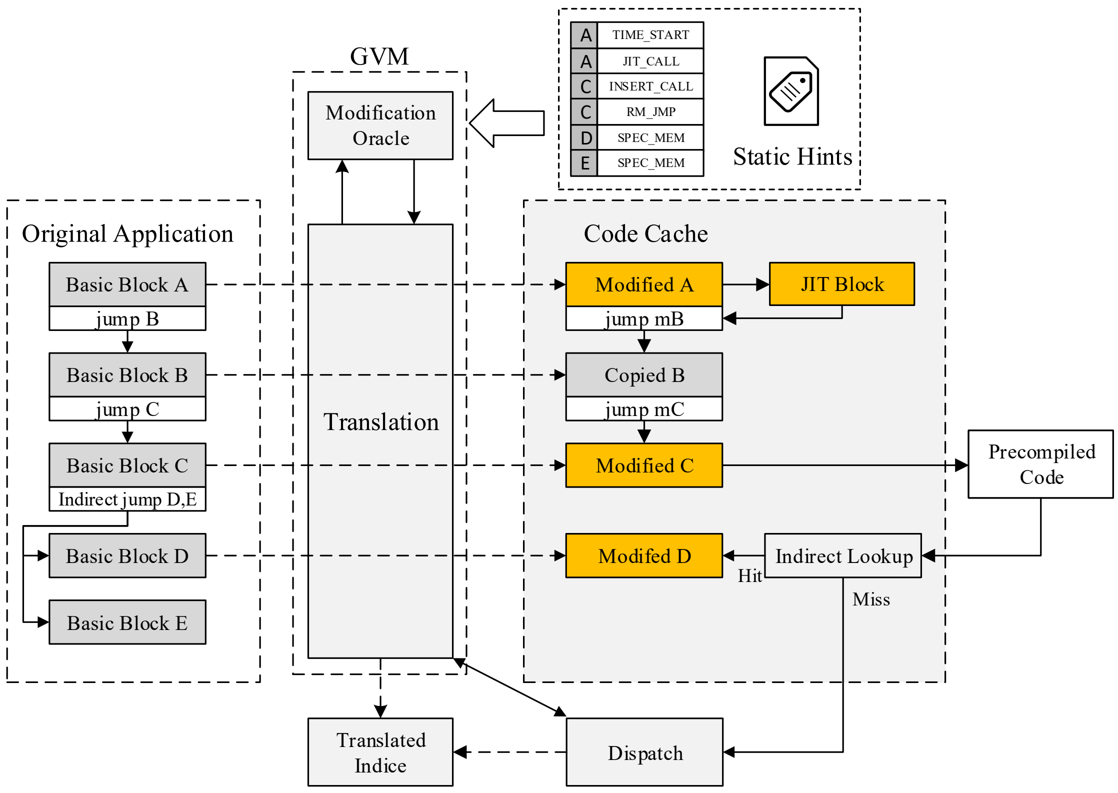


Figure 3.1: Dynamic binary translation queries the Modification Oracle to modify each basic block. The modification information is provided by static hint programs

can be arranged and controlled in a programmable, domain-specific language generated by the static binary analysis. For easier interpretation by the translator, the language can be designed as a sequence of *hint instructions*. Each hint instruction represents a simple modification pass that annotates a specific address with a dynamic JIT routine.

In this manner, the dynamic recompilation problem is transformed into two static problems:

- How to decompose the global recompilation problem into small modification passes.
- How to generate consistent hint instructions to control these modification passes.

The interpretation of hint instructions in the dynamic binary translator enables us to rethink the modification oracle as a *virtual machine*. In GBR, its modification oracle is called GVM (Guided Virtual Machine). GVM interprets hint instructions and performs patches on the “data”, which is the original instruction for each basic block. The hint instruction follows a custom *instruction set architecture* (ISA) and it is only recognised and interpreted by the GVM. The group of hint instructions are enclosed in a hint file called a *hint program*. The static binary analysis can be treated as a “virtual compiler” that compiles from the original executable and generates hint programs.

With the definition of hint instruction interface, the guided dynamic binary recompilation framework consists of two major components. One is the *static binary analyser* (SBA), which disassembles the target executable and generates hint programs. The other is the *dynamic binary recompiler* (DBR) that performs dynamic binary modification as specified in the hint program. Moreover, the GBR framework can also be optionally configured as a feedback system between SBA and DBR. The DBR could, in turn, generate profiling information to improve the accuracy of SBA and hint generation.

3.2 Design Choices

The implementation of a guided dynamic binary recompilation tool requires a significant amount of engineering work. Instead of re-inventing the wheel, a cost-effective solution is to reuse existing static/dynamic tools and libraries as much as possible. Additionally, to deliver information from static to dynamic in an automatic manner, an efficient interface between the static and dynamic tools must be built.

3.2.1 Dynamic Binary Translation

DynamoRIO [33] is chosen as GBR's dynamic binary recompilation platform. DynamoRIO is a robust enough and well-supported open-source DBT which originates from the well-known high-performance DBT: Dynamo [32]. It is robust enough to translate complex desktop-class applications such as web browsers, Microsoft Office, etc. As the hypothesis is to focus on binary optimisations for performance, the translation overhead must be minimised. Heavy-weight binary translation tools or instrumentation tools such as QEMU [25] and Valgrind [24] are more focused on cross-platform translation and instrumentation. They are not selected due to their high translation overhead.

Moreover, we aim to build a recompilation module in the DBT which requires transparent control of the application contexts and DBT internal code. Therefore another high-performance candidate, the Pin tool [22] from Intel is not preferred, since the Pin tool does not fully support fine-grained code transformation and it is difficult to manipulate JIT capabilities. Other light-weight DBTs such as DynInst [91] rely on inserting code snippets and leave other parts of code unmodified. Due to its limited code modification API, it is difficult to integrate JIT capability in these DBTs.

Among DBT tools, DynamoRIO is the most transparent platform to incorporate a JIT module to perform binary recompilation. It has the following strengths compared to other DBTs:

- Firstly, application contexts stay in the same register as specified in the original executable. This is a fundamental condition to guarantee the validity of information delivered from static binary analysis.

- Secondly, DynamoRIO makes it relatively easy to access runtime contexts and insert/modify/delete an instruction in a basic block. It maintains a cross-platform intermediate representation (IR) and has full encoding APIs to compile into native code.

- Thirdly, DynamoRIO achieves the best decoding and encoding efficiency [29]. In contrast to other runtime systems that convert all instructions to highly descriptive IR, DynamoRIO designs its IR as close to machine instructions as possible. It also performs a lazy decoding scheme where it only needs to decode when it needs to examine the instruction for modification. For most of the cases, it simply copies original byte codes to its code cache without performing decoding.

3.2.2 Static Binary Analysis

Despite the fact that there are many static binary analysis tools around, a custom designed static binary analysis tool is implemented for GBR. The prime consideration is to have static analysis that is aware of the nature and constraints of dynamic recompilation using DynamoRIO. For example, the analysis tool must have the same definition of data structures as DynamoRIO, such as IR abstraction level, basic blocks, control flow and heuristics. It is much easier to generate correct and compatible hint instructions that can be directly loaded and interpreted by DynamoRIO. Having a different definition in static analysis would cause unnecessary translation due to the incompatible data structure.

Secondly, there are many static analysis tools such as BAP [92], BitBlaze [93] and SecondWrite [94] that lift binary machine code to a much higher IR. From the rich context of high-level IR, they can perform heavy analysis by utilising existing analysis passes from compilers. However, for generating hint instructions, decompiling binaries to IR may result in a loss of mapping to the original hardware context allocation. For example, an original x86 instruction might be translated to one or more IR statements regarding its semantics, and flag manipulation, a register access or stack access might be both lifted to a uniform variable access as IR. While the design of hint instructions is intended to annotate on the original instructions, it is difficult to map the results of their analysis back to the original instruction.

As discussed in Section 2.2, accurate disassembly and control flow recovery is a difficult problem. The GBR static analysis aims to focus on analysis to generate hint instructions instead of focusing on CFG recovery. Therefore it is better to reuse existing static tools as the frontend for disassembly and CFG recovery. The Capstone disassembler library is selected as the frontend to process disassembling. For regular binaries, the GBR static analysis tool can recover most CFG while all indirect jumps are marked undetermined. For more accurate CFG information, it can load CFG files from the state-of-the-art commercial binary disassembler IDA Pro [95]. IDA Pro performs comprehensive data and control flow analysis even including interactions with users. However, the accuracy of CFG is not a necessity for GBR. Even with a fragmented and inaccurate control flow graph, GBR can still apply transformations on the parts of code that are proved accurate. For statically undecided control flows, it may generate extra runtime checks to maintain correctness.

3.2.3 Hint Instruction Interface

To enable efficient information delivery from the static binary analysis to the dynamic binary translation, the *hint instruction* is designed as a fixed length and RISC-like data structure. Raw bytes of hint instructions can be loaded into DBT as a whole without performing further decoding or decompression. Each hint instruction annotates a specific address with an opcode. Extra information of the modification contexts, such as the scratch register, stack operations or thread local storages, etc, are encoded in the "register" field of the instruction. Figure 3.2 shows an example of the hint instruction structure.

The hint instruction is similar to the DWARF [96] debugging information that annotates machine code with symbolic information. Similarly to generating debugging information, the static binary analyser can generate hint instructions according to standardised hint ISA. The details of the current implementation of the hint ISA are specified in Appendix B. The standardisation of the static-dynamic interface enables us to use different

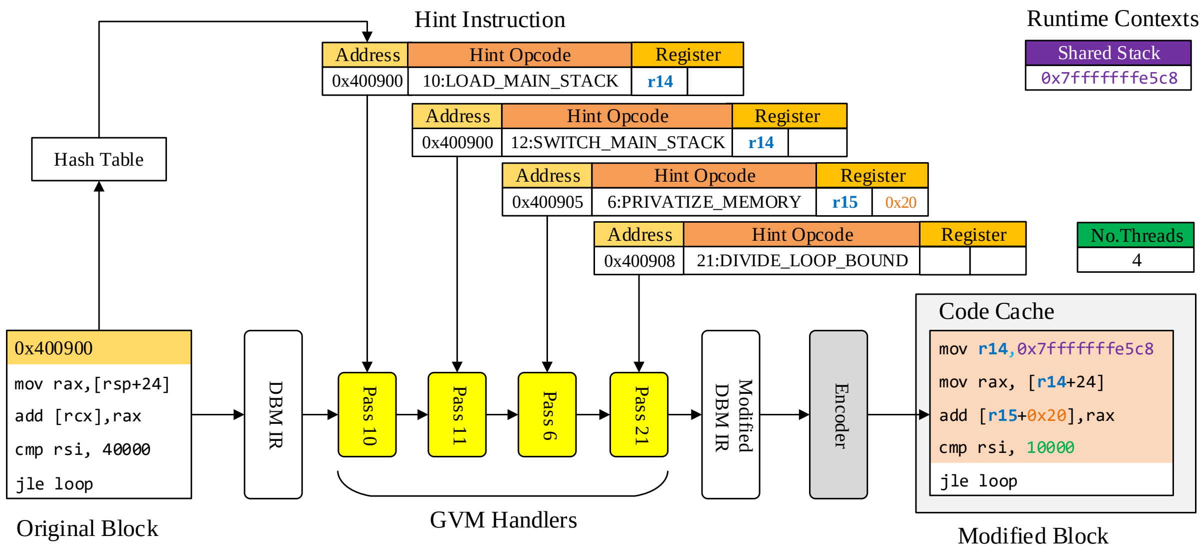


Figure 3.2: Hint instruction specifies opcodes and modification contexts for corresponding GVM handlers to instruct DBT to perform recompilation. The modification contexts provide information for thread local storage, liveness and flag information obtained from the static analysis.

implementations of static and dynamic tools while achieving compatibility for developments in the future.

The groups of hint instructions are encoded into a hint program. The hint program can be reused for recompilation as long as it is attached to the same executable. Besides hint instructions, the hint program contains a header that specifies the layout of the program. It specifies the global information of the executable such as the information of each function and loop. The global information is impossible to obtain in the DBT. With the information, the DBT can directly access the analysis results from static binary analysis immediately and efficiently.

3.3 Guided Binary Recompilation

This section discusses the implementation of the guided virtual machine (GVM) that interprets static hint instructions to perform comprehensive JIT recompilation. GVM is built as a client of DynamoRIO that is hooked in the basic block creation event. Figure 3.2 shows an example of the process. Before DynamoRIO copies each newly discovered basic block to its code cache, it looks up if there is a hint instruction associated with the block. If the translator finds the block has been tagged by a hint instruction and the runtime condition is true, it immediately performs modification by calling to the corresponding GVM handler specified by the hint instruction.

There are two different approaches to modification: the most typical approach is to patch the DynamoRIO IR representation of the block directly and let the DynamoRIO encoder compile into native code, as shown in Figure 3.2. The other approach is designed for a more complex modification—it dynamically replaces original application code with pre-compiled or JIT-complied code snippets.

The GVM consists of a group of pass handlers dedicated to achieving a fixed modification, such as redirecting control flows, replacing instruction operands or setting a dynamic flag, etc. Each pass handler is designed to perform a “patch” on the existing DynamoRIO IR. The patch is only effective once the input IR satisfies the handler’s designated run-time condition. As a whole, the GVM can be implemented as a simple switch system that redirects to the corresponding handler based on the opcode of each hint instruction. To add more functionality in the GVM, we can simply add new hint instruction opcodes to the GVM ISA and their corresponding handlers to the GVM.

3.3.1 Guided IR Modification

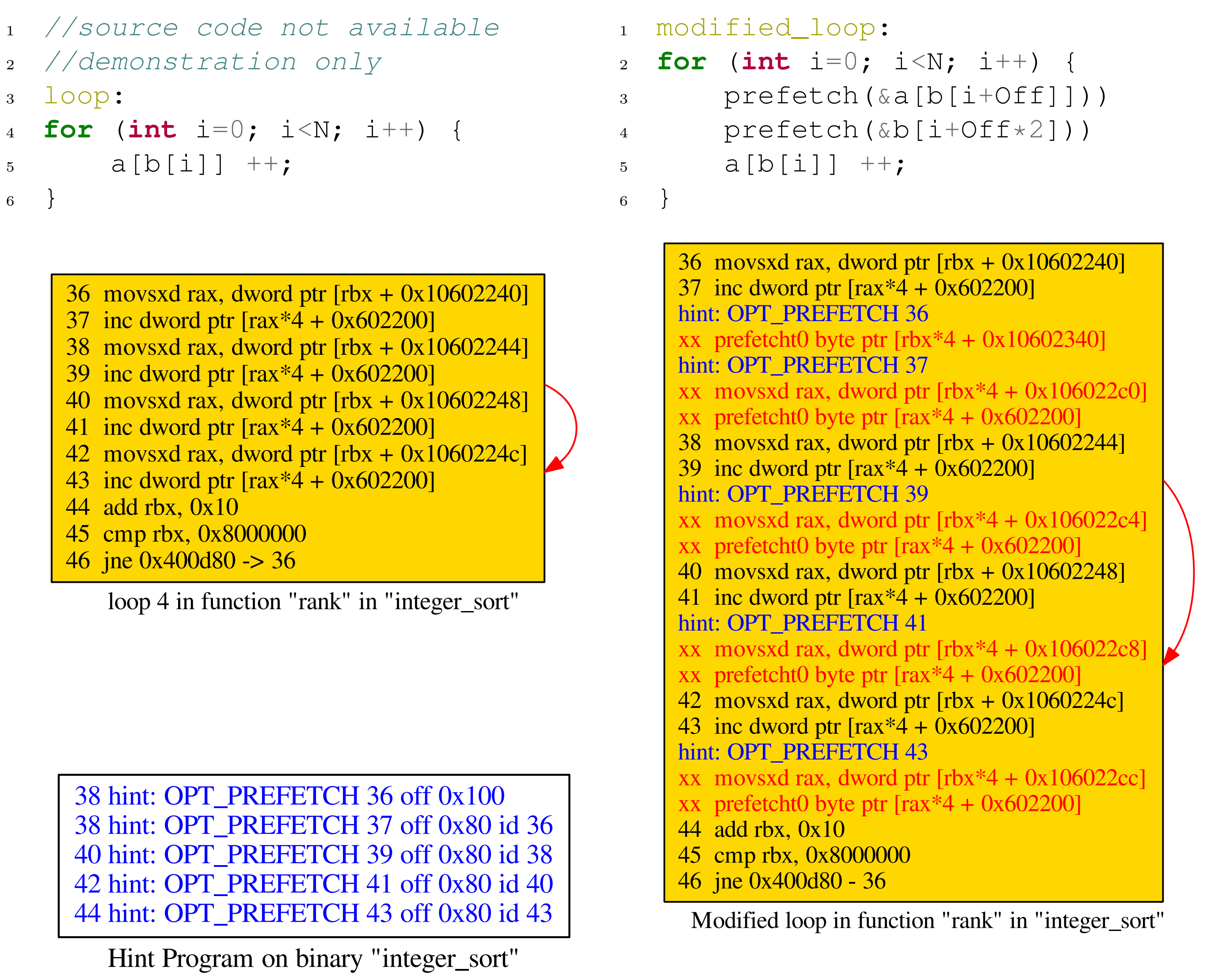
The combination of static binary analysis and dynamic binary translation offers enormous flexibility to modify binaries. The flexibility provides the opportunity to apply many existing optimisations to legacy binaries without access to source code. Instead of illustrating further implementation details, I present two optimisation case studies using the GVM. All examples demonstrate real recompilation performance on real systems without any hardware extension.

3.3.1.1 Case Study: JIT Prefetching Recompilation

One promising optimisation is automatic software prefetching. It is observed that many applications are heavily memory-latency bound, where memory latency has dominated the critical path and affected performance. Prefetching is a technique to bring the actual data from memory to fast caches before it is needed, which is an effective way to shrink the memory latency and improve performance. Conventional hardware is developed to prefetch regular memory access patterns [97] but could not handle irregular memory accesses. Software prefetching [98, 99] has been proposed to generate extra routines to predict the loading address based on knowledge of data structures and algorithms. However, the software prefetch technique is typically applied within compilers at source code level only. It is not applicable for legacy machine code.

This case study illustrates how the GBR recompilation framework brings performance improvement using the software prefetch optimisation on legacy executables. In this case study, we evaluate two examples: integer sort and hash join from the NAS benchmarks [100]. Figure 3.3 shows a typical loop that has recursive indirect memory accesses from the integer sort benchmark. For each iteration, it accesses different addresses of the array `a[]`, while the indices are purely determined by the data from array `b[]`. Indices from array `b[]` are typically discrete. It is likely a cache miss would occur when loading an element of array `a[]` at each iteration, meaning the processor has to stall its pipeline for fetching loads.

The prefetch optimisation is to insert extra software routines that load the data that would be read in iteration `i+offset` while it is still working on iteration `i`, where the `offset` represents the number of entries that can fit its data structure in a cache line. The recognition and transformation for software prefetch is implemented in the static binary analysis based on Sam Ainsworth’s work [99]. According to Ainsworth’s algorithm, the prefetch routine should prefetch the address of `a[b[i+Offset]]`. To further improve the performance, the address of the `b[i+2*Offset]` can also be fetched beforehand. The proposed loop after the software prefetch optimisation is shown in Figure 3.3 (top right).



```
//source code not available
//demonstration only
loop:
for (int i=0; i<N; i++) {
    a[b[i]] ++;
}
```

```
modified_loop:
for (int i=0; i<N; i++) {
    prefetch(&a[b[i+Off]]))
    prefetch(&b[i+Off*2]))
    a[b[i]] ++;
}
```

```
36  movsxd rax, dword ptr [rbx + 0x10602240]
37  inc dword ptr [rax*4 + 0x602200]
38  movsxd rax, dword ptr [rbx + 0x10602244]
39  inc dword ptr [rax*4 + 0x602200]
40  movsxd rax, dword ptr [rbx + 0x10602248]
41  inc dword ptr [rax*4 + 0x602200]
42  movsxd rax, dword ptr [rbx + 0x1060224c]
43  inc dword ptr [rax*4 + 0x602200]
44  add rbx, 0x10
45  cmp rbx, 0x8000000
46  jne 0x400d80 -> 36
```

loop 4 in function "rank" in "integer_sort"

```
36  movsxd rax, dword ptr [rbx + 0x10602240]
37  inc dword ptr [rax*4 + 0x602200]
hint: OPT_PREFETCH 36
xx  prefetcht0 byte ptr [rbx*4 + 0x10602340]
hint: OPT_PREFETCH 37
xx  movsxd rax, dword ptr [rbx*4 + 0x106022c0]
xx  prefetcht0 byte ptr [rax*4 + 0x602200]
38  movsxd rax, dword ptr [rbx + 0x10602244]
39  inc dword ptr [rax*4 + 0x602200]
hint: OPT_PREFETCH 39
xx  movsxd rax, dword ptr [rbx*4 + 0x106022c4]
xx  prefetcht0 byte ptr [rax*4 + 0x602200]
40  movsxd rax, dword ptr [rbx + 0x10602248]
41  inc dword ptr [rax*4 + 0x602200]
hint: OPT_PREFETCH 41
xx  movsxd rax, dword ptr [rbx*4 + 0x106022c8]
xx  prefetcht0 byte ptr [rax*4 + 0x602200]
42  movsxd rax, dword ptr [rbx + 0x1060224c]
43  inc dword ptr [rax*4 + 0x602200]
hint: OPT_PREFETCH 43
xx  movsxd rax, dword ptr [rbx*4 + 0x106022cc]
xx  prefetcht0 byte ptr [rax*4 + 0x602200]
44  add rbx, 0x10
45  cmp rbx, 0x8000000
46  jne 0x400d80 - 36
```

Modified loop in function "rank" in "integer_sort"

```
38 hint: OPT_PREFETCH 36 off 0x100
38 hint: OPT_PREFETCH 37 off 0x80 id 36
40 hint: OPT_PREFETCH 39 off 0x80 id 38
42 hint: OPT_PREFETCH 41 off 0x80 id 40
44 hint: OPT_PREFETCH 43 off 0x80 id 43
```

Hint Program on binary "integer_sort"

Figure 3.3: Transformation of integer sort from the NAS benchmark [100] to enable software prefetch. The transformation is decomposed into five hint instructions. Each hint instruction is responsible for one prefetch address.

Recompilation Workflow

The process of recompilation in GBR can be divided into two phases: static and dynamic. For the static phase, GBR performs static binary analysis on the input integer sort executable. It disassembles the executable, recognises loops and scans indirect array accesses from the recognised loops. Specifically for the recognition and transformation analysis of software prefetch, a custom analysis pass is implemented according to Ainsworth's algorithm. Figure 3.3 (middle left) shows a potential loop disassembly with indirect array accesses recognised by static analysis. As we can see, the loop was already optimised heavily by the original compiler with a loop unrolling factor of four.

To make the dynamic binary translator perform modifications for software prefetch, a hint opcode `OPT_PREFETCH` is inserted in the GVM virtual ISA, and a corresponding handler is implemented in the GVM. When each software prefetch opportunity is identified, the static binary analyser generates a hint instruction with the opcode `OPT_PREFETCH`.

Each hint instruction `OPT_PREFETCH` is responsible for modifying a single prefetch address at the specified location. It also specifies the prefetch offset from the annotated dynamic memory address. For example in Figure 3.3 (middle left), the dynamic memory address for instruction 36 is [`rbx` + `0x10602240`]. If the prefetch offset is calculated as `0x100` in the static analysis, the GVM prefetch hander would insert a prefetch instruction on [`rbx` + `0x10602340`] before instruction 36. Similarly, there are four other memory accesses that can be prefetched. Therefore five `OPT_PREFETCH` hint instructions are generated. They are encapsulated in a hint program, as shown in the bottom left of the Figure 3.3.

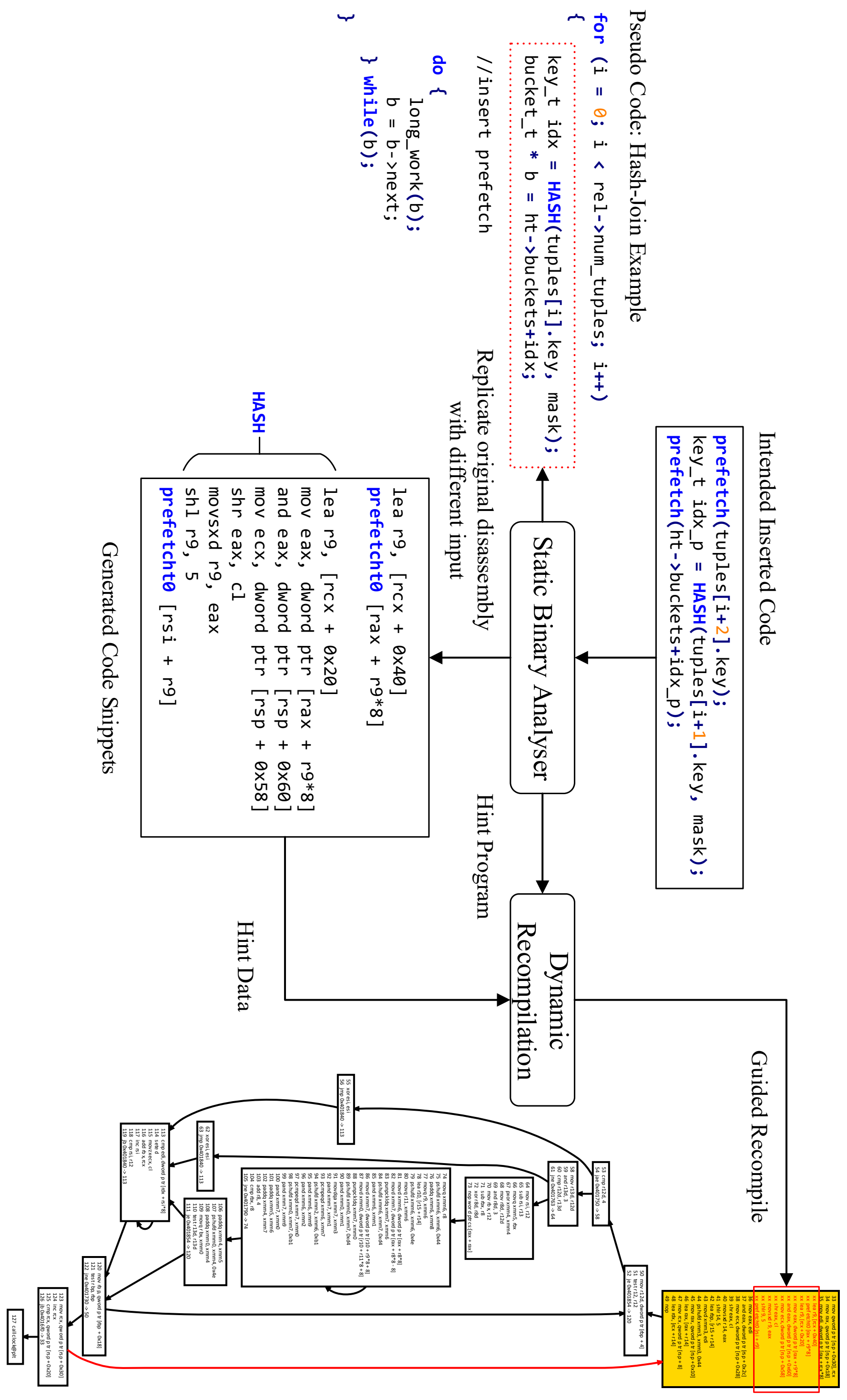
For the dynamic phase of GBR, the same executable is passed to run under DynamoRIO while loading all hint instructions from the hint program. When it reaches the basic block that has been annotated by the `OPT_PREFETCH` hint instructions, DynamoRIO invokes the GVM prefetch handler that interprets the five `OPT_PREFETCH` hint instructions respectively. Each invocation of the prefetch handler identifies the starting point for IR modification, validates the memory instruction, calculates the runtime prefetch address and inserts a new prefetch instruction based on the calculated address. After all hint instructions have been interpreted, the final modified IR for the block is passed to DynamoRIO's encoder to compile into native machine code, as shown in Figure 3.3 (right). The compiled byte code is then copied to the code cache and linked with other blocks as shown in Figure 3.1.

Complex prefetch routine generation

The advantage of software prefetching is to use additional information from data structures and algorithms to predict the next loading addresses. The software prefetch algorithm may perform additional computations to speculate the next access address. To demonstrate how GBR generates complex software prefetch routines, consider the Hash Join [101] benchmark. Hash Join is a kernel designed to simulate the behaviour of database systems. The pseudo code of Hash Join is shown in Figure 3.4 (leftmost). It maintains a loop where each iteration retrieves a bucket from a hash table lookup based on a key: `HASH(tuples[i].key, mask)`. From the bucket, it performs compute intensive searches in a linked list. To prefetch the bucket address of the next iteration, the bucket address should be calculated by calling the same hash function with the key of the next iteration: `HASH(tuples[i+1].key, mask)`.

Compared to analysis at compiler level, GBR does not have access to the source code. Data structure information such as the hash table construction is difficult to retrieve in the static binary analysis. A generic approach to identify the operations in data structures is beyond the scope of this dissertation. We leave the generalisation of the static recognition of data structures in binaries as a separate task. For the moment, the static binary recognition pass assumes the knowledge that the executable is a hash-join operation. It is hard-coded to scan hash table operations and recognise the corresponding location for prefetch modifications.

As shown in Figure 3.4 (top), the static binary analyser retrieves all related assembly of `HASH(tuples[i].key)` and replaces the input of the assembly with the next key: `HASH(tuples[i+1].key)`. Instead of copying the whole code of `HASH` function through the hint program, a hint instruction `REPLICATE_CODE` is inserted to specify the exact boundaries of the `HASH()` and insertion point. At the dynamic phase, when the GVM interprets the `REPLICATE_CODE` hint instruction, the code of `HASH()` is replicated



```
for (i = 0; i < rel->num_tuples; i++)
{
    key_t idx = HASH(tuples[i].key, mask);
    bucket_t * b = ht->buckets+idx;

    //insert prefetch

    do {
        long_work(b);
        b = b->next;
    } while(b);
}
```

```
prefetch(tuples[i+2].key);
key_t idx_p = HASH(tuples[i+1].key, mask);
prefetch(ht->buckets+idx_p);
```

```
lea r9, [rcx + 0x40]
prefetcht0 [rax + r9*8]

lea r9, [rcx + 0x20]
mov eax, dword ptr [rax + r9*8]
and eax, dword ptr [rsp + 0x60]
mov ecx, dword ptr [rsp + 0x58]
shr eax, cl
movsxd r9, eax
shl r9, 5
prefetcht0 [rsi + r9]
```

Figure 3.4: Software prefetch transformation of the Hash Join [101] example. The static binary analyser retrieves all related assembly of `HASH(tuples[i].key)` and replaces the input of the assembly with the next key: `HASH(tuples[i+1].key)`. The details of code replication is performed at runtime guided by the hint instruction.

with a new argument `tuples[i+1].key`, as shown in 3.4 (right). The rest of the compute intensive code remains unchanged throughout the dynamic translation.

Evaluation

To demonstrate the effectiveness of the JIT recompilation framework, this section evaluates the JIT optimisation performance for the above two examples: integer sort (IS) and hash join (HJ). In this experiment, executables of IS and HJ are requested from Ainsworth without asking for the source code. They stand for unoptimised legacy executables that were compiled for old machines without prefetching capabilities (without instruction `prefetcht0`). I also requested pre-compiled executables that are compiler generated and optimised for software prefetching to compare the performance difference between JIT and pre-compiled code. The experiment was performed on a Sandy Bridge Intel(R) Xeon(R) CPU E5-2667 @ 3.30GHz processor. The processor has 32KB L1 cache and supports the `prefetcht0` instruction. The `prefetcht0` fetches the line of data from memory into all levels of the cache hierarchy including L1. The whole experiment ran in a fully automatic manner without manual intervention.

Figure 3.5 shows the mean relative execution time from ten runs. The DBT unmodified refers to running the original executable under dynamic binary translation without performing any modification. It reflects the overhead of dynamic binary translation. For the two benchmark programs, GBR can achieve translation performance that is close to native execution. This is because most of the execution time is spent in the code cache with translated fragments linked directly. As there are few indirect jumps for both benchmarks, the binary translation overhead is considered negligible. For benchmark HJ, GBR's DBT even outperforms the native execution due to trace optimisation.

With the low translation overhead, the guided JIT recompilation demonstrates a significant performance improvement through performing software prefetching modifications on the original executables. For integer sort, the JIT-recompiled performance is close to pre-compiled. As the JIT modification results in the same code as the pre-compiled version, the gap between the JIT-compiled and Pre-compiled represents the overhead of the JIT recompilation.

For the hash join benchmark, it is found that the JIT recompilation even out-performs the pre-compiled version. I examined the JIT modified code and found that the performance gap was not due to software prefetching, but was caused by inlining optimisation. The original executable was generated by heavy compiler optimisations where both the `HASH` function and the containing function are inlined to their parents. While in the pre-compiled HJ executable, we found the hot loop with the prefetched instruction is not inlined. As the routine is in a frequently executed region, the extra call and return instructions would cause a performance penalty. The performance gap reveals another big advantage of JIT-recompilation, it can perform independent post-optimisations without interfering with all previous compiler optimisation passes. While for the compilers, inserting an extra optimisation pass in the IR, such as software prefetch optimisation pass, may confuse the inlining pass and affect the ability to further inline functions during compilation.

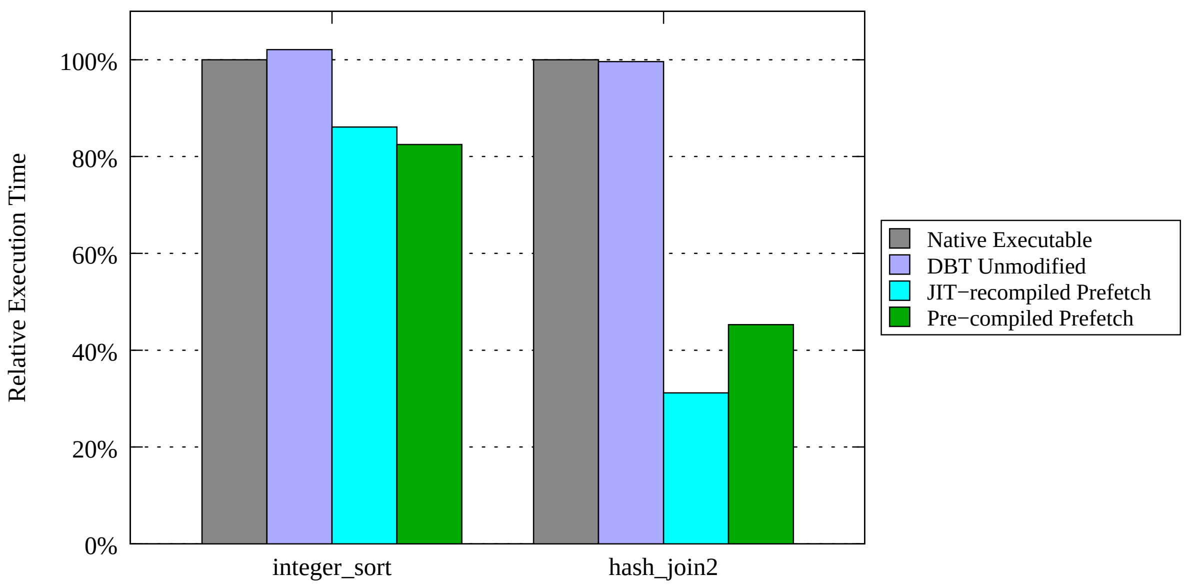


Figure 3.5: Software prefetch performance comparison between the JIT-recompiled and pre-compiled executable.

3.3.2 Guided Binary Instrumentation

Besides directly supporting DynamoRIO IR modification, GBR also supports loading, replacing and linking code from pre-compiled objects or recompiled code from other compilers. Runtime linking is not a new concept and it is widely used in many aspects: some runtime linking is performed actively by the executable itself, such as dynamically linking a shared library using the global offset table (GOT). Other linking techniques are conducted passively through dynamic binary translation, such as binary instrumentation and profiling applications [33].

Typical binary instrumentation techniques are based on inserting call instructions or trampoline jumps to user instrumentation functions. Conventionally runtime information such as register and memory accesses, basic block boundaries, function calls are accessible and examined purely in a dynamic binary translator. However, for runtime events that are related to data structures, types and algorithms require the knowledge of global contexts, which are not easily detected dynamically. Due to this limitation, compiler generated instrumentation is preferred over binary instrumentation, due to its access to the extra symbolic information from the source and global view of the programs. However compiler-based instrumentation has its own disadvantage: it is very language-dependent and lacks the ability to monitor the compiled machine contexts.

Some DBT platforms [33, 22] seek to strengthen binary instrumentation with additional static analysis integrated into the DBT, such as retrieving symbol tables or liveness analysis. For example, the latest version of DynamoRIO also supports static user annotation on binaries so that they can be loaded at runtime. However performing a global analysis at runtime is rather expensive.

Through the hint instruction interface, GBR enables a high-level and user-friendly interface to control binary instrumentation. The static binary analysis is implemented as a library so that users can write a simple client to examine the binary globally and hierarchically. Figure 3.6 illustrates an example structure of user instrumentation hint insertion. Statically, users have the freedom to navigate and examine each instruction in

```
for (auto function : module.functions) {
  for (auto loop : function.loops) {
    for (auto block : loop.blocks) {
      for (auto inst : block.machineInsts) {
        if (inst.writesInductionVariable(loop))
          insertHint(inst, INDUCTION_READ);
      }
    }
  }
}
```

Figure 3.6: Examples in writing a static client that uses global static information to guide binary instrumentation for a specific purpose.

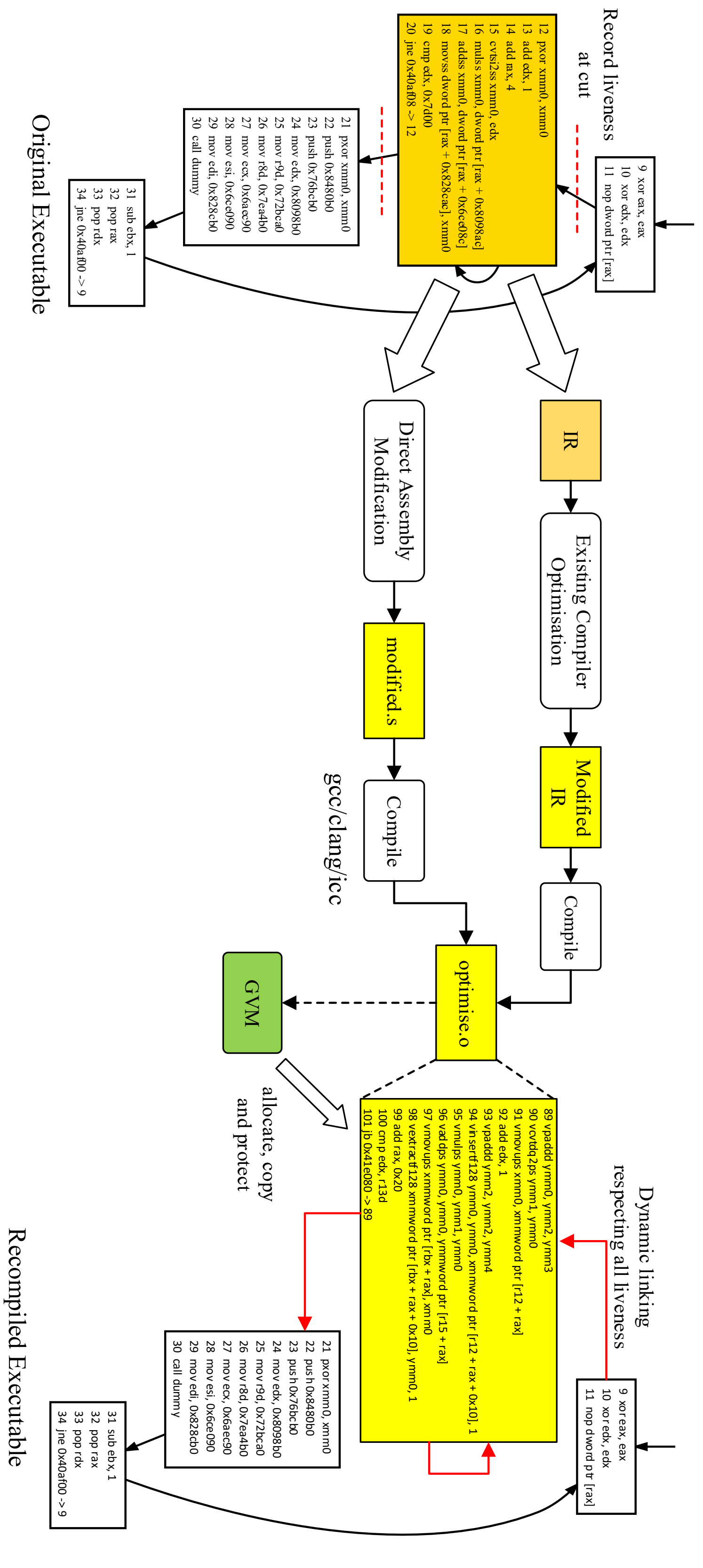
the executable, using existing heavy-weight static analysis algorithms and insert hints at places of interests.

Dynamically, each hint instruction is appointed with a user-defined handler so that a particular user instrumentation function or list of dynamic inlined instruction listed can be inserted at places of interest. The instrumented code is then executed when the runtime reaches to the hint annotated location. Compared with other binary instrumentation platforms to runtime filters or samplers, GBR offers a platform to control the instrumentation context ahead of time through static analysis. The scope of instrumentation was significantly expanded. More details about the application of hint instructions to guide sophisticated binary instrumentations will be discussed in chapter 4.

3.3.3 Partial Static Recompilation

Similar to binary instrumentation, the linking to pre-compiled code can also be used for performance optimisation. A fraction of the executable, especially for the frequently executed region, can be completely replaced with more efficient code for better performance. This requires the prior knowledge of the "hot" code that are frequently re-executed through dynamic coverage analysis. If the hot code consists of one or multiple basic blocks that can be thoroughly analysed statically, it is more cost-effective to recompile the blocks statically. The dynamic binary translator can link the recompiled code snippet at runtime and execute the code natively without copying it to the code cache. For hot regions whose control flow or data flow cannot be determined statically, they are not applicable for static partial recompilation.

Figure 3.7 shows the process of partial static recompilation. For the first pass, GBR needs to profile the executable to locate the repeatedly executed regions with high coverage in the executable. At the second pass, the profiling information is sent to static analysis and liveness analysis is performed on each hot code block to ensure it is applicable for static partial recompilation. Once the hot code is verified and retrieved, two approaches are used to optimise the hot code. One is to directly modify the assembly if trivial optimisation opportunities such as loop unrolling or loop code motion are observed. For a more complex modification, the hot machine code is lifted to a compiler-compatible IR such as the LLVM IR, using existing binary decompilation libraries such as McSema [102]. From the IR level, all existing conventional compiler transformations are then applicable to be used for optimisation.



```
9  xor eax, eax
10 xor edx, edx
11 nop dword ptr [rax]
```

```
12 pxor xmm0, xmm0
13 add edx, 1
14 add rax, 4
15 cvtsi2ss xmm0, edx
16 mulss xmm0, dword ptr [rax + 0x8098ac]
17 addss xmm0, dword ptr [rax + 0x6ce08c]
18 movss dword ptr [rax + 0x828cac], xmm0
19 cmp edx, 0x7d00
20 jne 0x40af08 -> 12
```

```
21 pxor xmm0, xmm0
22 push 0x8480b0
23 push 0x76bcb0
24 mov edx, 0x8098b0
25 mov r9d, 0x72bca0
26 mov r8d, 0x7ea4b0
27 mov ecx, 0x6aec90
28 mov esi, 0x6ce090
29 mov edi, 0x828cb0
30 call dummy
```

```
31 sub ebx, 1
32 pop rax
33 pop rdx
34 jne 0x40af00 -> 9
```

```
9  xor eax, eax
10 xor edx, edx
11 nop dword ptr [rax]
```

```
89 vpaddd ymm0, ymm2, ymm3
90 vcvtdq2ps ymm1, ymm0
91 vmovups xmm0, xmmword ptr [r12 + rax]
92 add edx, 1
93 vpaddd ymm2, ymm2, ymm4
94 vinsertf128 ymm0, ymm0, xmmword ptr [r12 + rax + 0x10], 1
95 vmulps ymm0, ymm1, ymm0
96 vaddps ymm0, ymm0, ymmword ptr [r15 + rax]
97 vmovups xmmword ptr [rbx + rax], xmm0
98 vextractf128 xmmword ptr [rbx + rax + 0x10], ymm0, 1
99 add rax, 0x20
100 cmp edx, r13d
101 jb 0x41e080 -> 89
```

```
21 pxor xmm0, xmm0
22 push 0x8480b0
23 push 0x76bcb0
24 mov edx, 0x8098b0
25 mov r9d, 0x72bca0
26 mov r8d, 0x7ea4b0
27 mov ecx, 0x6aec90
28 mov esi, 0x6ce090
29 mov edi, 0x828cb0
30 call dummy
```

```
31 sub ebx, 1
32 pop rax
33 pop rdx
34 jne 0x40af00 -> 9
```

Figure 3.7: Frequently executed region in the original executable is retrieved, recompiled statically and linked back at GBR runtime. The assembly code shows an example of a loop compiled for the SSE extension to the new AVX2 extension.

From either approach, the transformed code can be compiled back to machine code and encoded into a code snippet. At the dynamic phase of GBR, the new code snippet is loaded into GVM's code cache. GVM marks the newly-loaded code with the permission of read-only, executable and write protection. An extra hint instruction is annotated on the designated cut location during the hint generation stage so that at runtime, when the application reaches to the cut boundary, GVM can interpret hint instructions to directly jump to the new efficient recompiled code snippet.

3.3.3.1 Case Study: Binary Vectorisation

The second case study demonstrates the feasibility of static partial recompilation by supporting vectorisation transformations on binaries. Since 1976, vector (SIMD) execution has gained increasing popularity, especially for applications in the signal-processing and scientific-computing domains. These vector instructions not only provide energy efficiency but also better performance through exploiting fine-grained data parallelism. Over two decades, SIMD hardware has been extended in both capacities and flexibilities. For example, x86 SIMD registers evolved from 128-bit (SSE) to 512 bits currently (AVX-512). More conditional and speculative executions are still being added in the x86 ISA. However, there exist a lot of legacy executables which are not compiled for vector acceleration. Also there are legacy compilers for old languages (e.g. Fortran77) which don't support vectorisation extensions.

To demonstrate that performance can be achieved through partial static recompiling with SIMD instructions, I give an example from the *Test Suite for Vectorizing Compilers* (TSVC) benchmark [103] which was developed to assess the vectorisation capabilities of compilers. The benchmark contains 151 typical small loops with different scenarios for vectorisation. The legacy executable to be vectorised is called `novec` compiled by `gcc` with `-O3` optimisation flag without any vectorisation flag. As for comparison, a pre-compiled vectorised TSVC executable `vec` is also prepared. Since this case study is not to illustrate vectorisation algorithms, we selected three representative loops from them for recompilation while using the same vectorisation algorithm. The assembly for three selected loops is retrieved from the GBR static analysis and directly transformed into the similar vectorised assembly shown in Figure 3.7.

For each loop, two versions of transformation are generated, one is targeting for x86 architecture extension `sse4.2` with 128-bit SIMD lanes. The other is for the more recent extension `AVX2` instruction set with lanes of 256-bit width. During the dynamic phase of GBR, it detects the available hardware SIMD extensions of the underlying CPU. For different hardware, GBR can recompile from the same original binary and adapt to the running hardware.

Figure 3.8 illustrates the performance difference between the JIT-vectorised from GBR for SSE4.2 (128-bit SIMD), AVX2 (256-bit SIMD) and the pre-compiled vectorised executable for SSE4.2. The experiment was performed on the Intel(R) Core(TM) i5-4670 CPU @3.4GHZ. From the results, we can see that the GBR JIT performance can achieve a similar performance improvement compared to the pre-compiled version. Again the overhead from dynamic binary translation is negligible. It demonstrates that performance can be still achieved without the need for source code. Moreover, GBR demonstrates its strength in dynamic adaptation and out performs a static pre-compiled executable. The same legacy executable can be dynamically recompiled based on the hardware features. It is extremely helpful for optimising legacy software when new generations of hardware

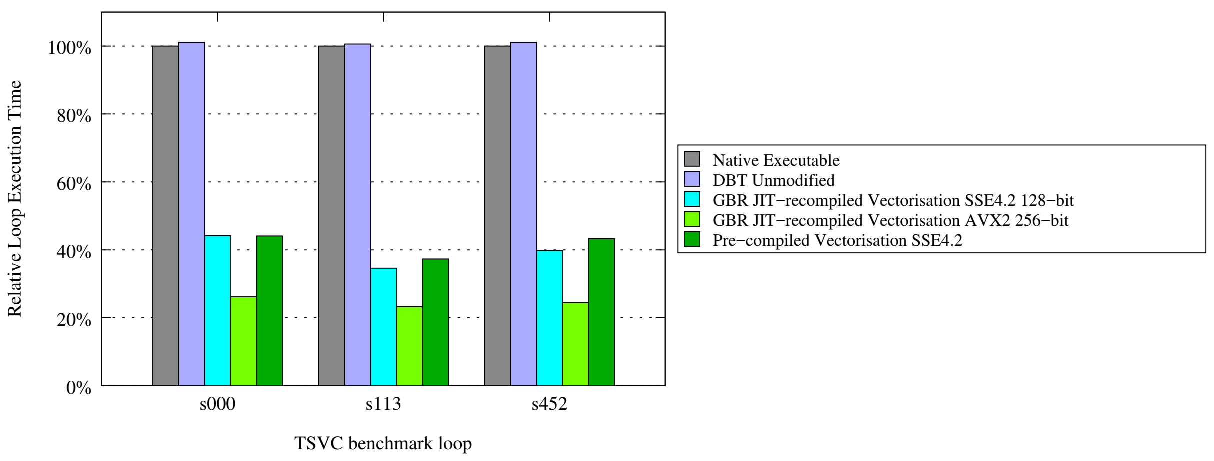


Figure 3.8: Vectorisation performance comparison between the JIT-recompiled and pre-compiled executable on three selected TSVC benchmark loops. From the same executable, GBR recompiles the loop into two different versions based on the available extension of the hardware (SSE4.2 or AVX2).

are released.

3.4 Related Work

Tools for binary modification and recompilation have been studied for years. Although many DBT tools allow users to perform customised modification for different purposes such as architectural compatibility, instrumentation, emulation and optimisation [33, 22, 104, 105, 106], few are related to the automated process of recompiling generic executables.

GBR is built on top of DynamoRIO [33], it inherits all the advantages in dynamic binary modification from DynamoRIO in terms of transparency, performance and flexibility. GBR integrates a virtual machine GVM that interprets hint instructions to perform modification on DynamoRIO IR, which augments DynamoRIO's ability to perform automatic recompilation. However, I am not aware that there are tools that offer the same programmable and flexible approach as GBR, to automate and control the sets of consistent and fine-grained modifications on application binaries. In this section, I list related static or dynamic tools that serve the same objective or functionality as GBR.

To my knowledge, the combination of Calpa [107] and DyC [108] are the most similar to our framework. DyC is a JIT compiler driven by user annotations on C source programs. The annotation specifies an intermediate structure for variables and code to be lazily compiled to the underlying hardware at runtime. Performance can be achieved through polyvariant specialisation[1], dead code elimination and dynamic peephole optimisations, such as strength reduction. As manually inserting annotations is difficult and error-prone, Calpa is designed as an automation tool that analyses source code and profiling information to generate correct and cost-effective annotations to guide DyC dynamic compilation. To use Calpa, a programmer firstly needs to run its profiling tool to instrument the application. The results from the profiling run and the applications original C source are then analysed by Calpa's program analysis tool, which automatically produces

[1]Depending on the value of a static variable, different versions of efficient code can be generated. The technique is typically used in dynamic loop unrolling

annotated C code for the application. Compared to GBR, Calpa-DyC is limited in annotating C languages while GBR is able to annotate on generic binaries compiled from different languages. Regarding dynamic optimisation, DyC was mainly implemented for the polyvariant specialisation optimisation, but GBR can be configured to more purposes such as parallelisation, prefetch and vectorisation.

The Sun Studio Binary Code Optimizer [109] and Microsoft Vulcan [105] & BBT [110] are two well-known commercial tools for rewriting binaries for better performance. Both tools can rewrite binaries without source code but rely on instrumenting the binary with training inputs to collect profiling information as the first pass. Both tools statically rewrite the binary and achieve significant performance with the help of profiling information, code layout heuristics and data flow static analysis. The rewriting is typically adding new segments along the original binary and replaces the original hot code with new efficient code sections. The benefit of rewriting binaries statically is that it incurs no dynamic overheads. However, it is limited to modifying binaries with simple control flows. The optimisation algorithms for both tools are more focused on single-threaded cache performance and control flow optimisation. Due to the limitation of static rewriting, they failed to perform more aggressive transformations such as parallelisation or vectorisation.

ADAPT [111] is a dynamic compiler and optimiser that rely on heuristics to apply runtime optimisation on C language. Instead of using annotation, users describe modifications in a domain specific language. The domain specific code includes information including runtime conditions to select the best version from different pre-compiled versions. ADAPT is built on top of Polaris [56] and is also able to perform transformations to enable parallelisation at source level. Compared to GBR, ADAPT relies on users to write heuristics to perform optimisation. This approach is limited in its applicability due to the extra burden it incurs on the user to manually annotate the source program.

3.5 Summary

This chapter presents a binary recompilation framework called GBR. GBR performs static binary analysis and uses scalable and programmable hint annotations to direct fine-grained dynamic modifications on general binaries. Compared to other automation tools for binary modification, GBR can express complicated optimisation operations in a domain-specific hint program. Therefore, it enables many state-of-the-art optimisation opportunities that were previously not applicable for legacy binaries or executables without source code. Through two case studies of applying software pre-fetching and vectorisation optimisations in GBR, it demonstrates many advantages brought by guided dynamic recompilation:

- It achieves substantial performance improvement on real CPU systems.
- It resolves the lack of global understanding of programs in dynamic translation through static binary analysis.
- It can access runtime information and partially resolve the ambiguity issues in static binary analysis.
- It performs dynamic optimisations such as trace optimisation so that the dynamic modification overhead is amortised.

- It provides optimisations that are independent of other compiler optimisation passes. Therefore it may achieve better performance compared to pre-compiled binaries with the same optimisation.

- It dynamically recognises the availability of hardware extensions and generates JIT code from the same executable to adapt to the underlying hardware.

Therefore it proves the first research hypothesis in this dissertation: a combination of static binary analysis and dynamic binary translation can bring efficient binary recompilation and achieve better performance compared to the native execution of the original binary.

GBR is designed as an open platform for binary recompilation. Custom analysis passes and hint generation can be inserted in the GBR static binary analysis. And the corresponding dynamic pass can be implemented in the GBR dynamic binary translator. However for more aggressive optimisations, the accuracy of the static analysis significantly affects the ability for GBR to recognise opportunities for optimisation and generate correct hints. Currently, the static binary analysis for GBR is still at its initial stage, it is not able to retrieve information as data structures or high-level semantics from the executable. Some implementation still lacks the ability to fully automate the hint generation process. In the future as more existing decompilation and analysis components are integrated into the static analyser, it is possible to implement more comprehensive analysis so that automatic hint generation can be achieved.

GBR is the fundamental infrastructure to enable further research for this dissertation. In Chapter 4, I will use GBR as a binary instrumentation platform to investigate parallelism in the general executables. Chapter 5 extends GBR with threading to implement an automatic paralleliser that recompiles binaries for concurrent execution.

Chapter 4

Uncovering Parallelism In Binaries

Building on the recompilation opportunities provided from the guided binary recompilation tool (GBR), the next step is to apply existing optimisations on legacy binaries. One of the most ambitious and challenging optimisations is automatic parallelisation. It extracts thread-level parallelism and transforms sequential binaries for parallel execution in an automatic manner.

This chapter investigates the feasibility of recognising and implementing parallelisation for legacy binaries through a limit study of parallelism in binary executables. Despite there already being many related parallelism limit studies (discussed in Section 4.4), at machine code level, it remains uncertain where and how to recognise parallelism and effectively extract it to execute on real systems. The preliminary limit study is essential for planning before actually implementing a full binary paralleliser.

As most of the program execution time is spent in loops, this chapter focuses on loop-level parallelism from general binaries. For a given application machine code, it divides the binary parallelism study into three problems for investigation:

- How to retrieve sufficient information from binaries for parallelism limit studies. Specifically, how to obtain accurate data and control dependence relations from machine code.
- What is the theoretical upper bound of parallelism based on the dataflow and control flow nature from the machine code.
- What is the estimated speedup given different realistic assumptions of parallelisation paradigms and real hardware under conservative static analysis.

4.1 Demand-driven Instrumentation

For the first problem, the exact control and data dependence information is typically retrieved by profiling through dynamic binary instrumentation. However, a pure dynamic binary instrumentation has a few disadvantages. Firstly, it is not scalable as it suffers from runtime and memory overhead when profiling large and long-running applications. Secondly, it lacks the control to filter out unnecessary runtime information and only examine runtime features that are of interest. Thirdly, it lacks the global understanding of the binary. High-level information such as data structures could not be easily obtained from binary instrumentation. Therefore the alternative approach: compiler-based instrumentation is more popular compared to binary instrumentation if the source code is available.

To address the disadvantages, a combination of static binary analysis and binary instrumentation can be used through the GBR platform as shown in Figure 4.1 (left). By using the GBR hint programs, fine-grained instrumentation operations can be controlled by the static binary analysis, as discussed in Section 3.3.2. Specifically, it performs static analysis on binaries and generates "questions" that are encoded in hint instructions. The questions are then answered by the dynamic binary instrumentation in its runtime event handlers. We call this type of binary instrumentation *Demand-driven Instrumentation*.

Demand-driven instrumentation has a few advantages compared to normal binary instrumentation. Firstly, data dependence relations that are fully decided through static analysis can be elided from dynamic profiling. Only the statically-undecided memory accesses are passed to dynamic binary instrumentation. A large fraction of unnecessary runtime information can be filtered out and thus reduce runtime and memory overheads. Secondly, global information of the program can be delivered through hint programs by matching runtime accesses from high-level data structure information such as induction/reduction variables, array or linked-list accesses. Consequently, it supports the same comprehensive instrumentation as the compiler-based instrumentation while the instruction-level information is still accessible. Thirdly, different hint programs can be generated from the same executable for different instrumentation purposes, such as coverage analysis, dynamic call graph analysis, dynamic data dependence analysis, etc.

4.1.1 Binary Emulator For Estimating Parallelism

To support binary implementation for evaluating the potential of enabling automatic parallelisation, GBR is extended to support demand-driven binary instrumentation. A list of instrumentation related hints are added in the hint ISA and many instrumentation handlers are added in GVM. The detailed hint opcodes for instrumentation can be found in the Appendix B.

The entire GBR extension to enable demand-driven instrumentation for evaluating parallelism is called **Binary Emulator for Estimating Parallelism**(BEEP). BEEP takes a standard executable along with the executable's training inputs and specified static loop id for instrumentation. It then instruments the executable once and outputs a detailed report on the potential of loop parallelism. The report concludes estimated speedup under different parallelisation assumptions. In addition to the report, it also generates hint information which could assist static binary analysis for actual parallelisation, which is discussed in chapter 5.

BEEP runs the application in single-threaded mode, but it emulates a hypothetical multi-threaded execution in the background. As shown in Figure 4.1, it consists of three major components: a *static binary analyser*, a *binary emulator* and many *parallel execution models*. The binary emulator is implemented in GVM to interpret hint instructions and generating runtime instrumentation *events* from the runtime instruction stream. The dynamic events such as register accesses, memory accesses, basic block boundaries, loop iterations, function calls can feed *parallel execution models* for evaluating parallelism. Note that some runtime events are not easily detected dynamically. For example, the exact point when a loop iteration starts, or instructions that access induction variables from a loop, cannot be easily found dynamically without a complex runtime sampler and

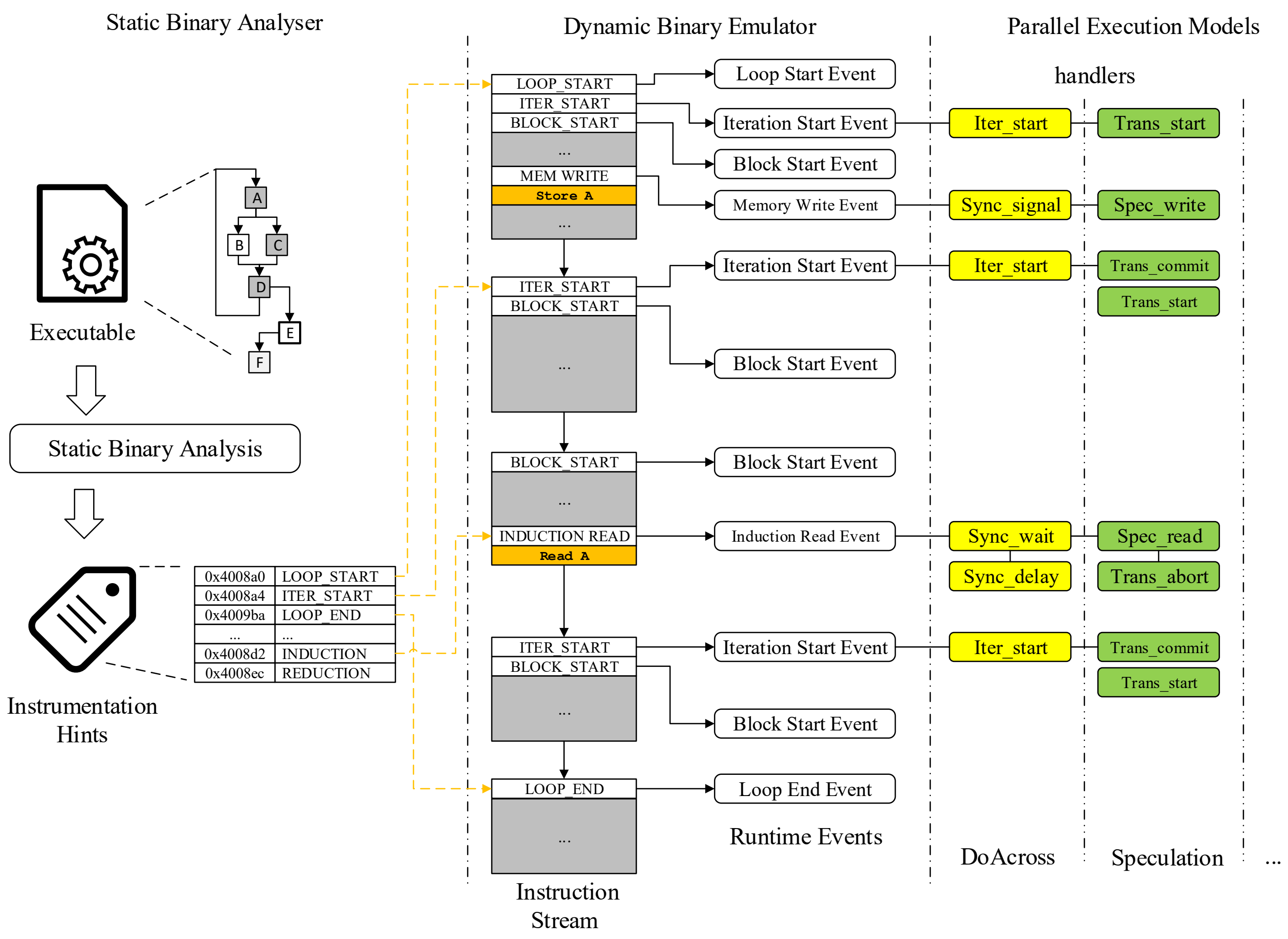


Figure 4.1: BEEP static binary analyser generates hint programs to control fine-grained processes of binary instrumentation to feed information to be evaluated in parallel execution models (right).

profiler. Therefore, BEEP has to rely on the hints from the static analysis. With the global static view of the whole binary, it can generate high-level events on data structures and control-flow paths which the dynamic tool could not simply discover.

With these runtime event callbacks that can be manipulated statically, we could build many abstract parallelisation execution models on top of BEEP binary emulator. *Parallel execution models* are a set of models that handle dynamic runtime events differently. For example, the models receive a stream of read, write addresses and loop events from the binary emulator. Based on the access order, the dynamic dependence graph could be built in the parallel model to evaluate parallelism using different paradigms.

Parallel execution models can also abstract underlying parallel machines with parallelisation hypotheses, to investigate parallelism benefits under the assumption of communication costs or hardware constraints. Multiple parallel models could be built independently and run together. Each model maintains a private array of cycle counters to record cycle timestamps of each runtime thread event without interfering with other models. Based on the model specification to handle data dependencies, delays may be added to correctly model compute or data transfer latencies. The result of a parallel model is typically an estimated speedup for the given assumption.

While much effort has been made to implement an efficient binary instrumentation tool to evaluate parallelism, this chapter is more about retrieving parallelism from the executable instead of the implementation techniques. In the next section, I will discuss a list of parallel execution models, their results and how they expose parallelism based on their assumptions. A fraction of the parallel execution models is shared and was contributed by the automatic parallelisation group led by Timothy Jones in the Computer Lab. Early concepts and results of some analysis components were also presented in Niall Murphy's work [112].

4.1.2 Benchmarks

To investigate parallelism opportunities for general applications, the SPEC CPU2006 (or SPEC2006) benchmark suite[113] was selected for investigation. SPEC2006 covers a wide range of computing intensive workloads developed from a variety of practical user applications. Table 4.1 shows the list of applications investigated in this dissertation. SPEC2006 was chosen over other scientific, media or streaming benchmark suites because it represents a more generic spectrum of applications which might contain irregular patterns and they are considered difficult to parallelise. All the executables are compiled by `gcc -O3` with maximum optimisation for single threaded performance. The SPEC2006 executables are treated as legacy binaries and suitable candidates for instrumentation without the help of source code.

4.1.3 Loop Coverage Profiling

Given a large pool of complex and obfuscated legacy executables, the first problem is to locate the most loop parallelism quickly. A natural way to locate most parallelism is to find the regions that are repeatedly executed and with high coverage over the whole execution time. For simplicity, we assume each instruction has a latency of one cycle. Therefore, the *coverage* of a loop is calculated as the total number of dynamic instructions from all invocations of the loop over the total number of dynamic application instructions. Instructions from sub-functions, sub-loops and shared library calls in the loop are also counted. The coverage is an accumulation of factions from all loop invocations. It means small loops with a high number of invocations are also considered as high-coverage loops.

The loop coverage information can be obtained from profiling using the demand-driven binary instrumentation in BEEP. From the static binary analysis, it annotates the start and finish addresses of each recognised static loop using hint instructions. At runtime, each static loop is allocated with a dynamic counter. When it reaches the annotated basic block of a loop, it enables or disables the loop's corresponding counter. Later for each incoming basic block, it increments all the enabled loop coverage counters by the size of the basic block. After completing the application, the coverage of each loop can be calculated by the ratio of its loop counter with the total global application instruction counter.

All loops from each SPEC2006 executable listed in Table 4.1 are assigned with a loop ID. Multi-dimensional loops and nested loops are treated as different loops with different loop IDs. For each loop, its coverage is profiled with training inputs. Profiling on a single training input may not represent the true application execution for all inputs. Therefore, each SPEC2006 application is profiled with different training inputs respectively. The

Benchmark	Binary Size	Original Language	Application Area	Description
Integer Benchmarks				
400.perlbench	1.5 MB	C	Programming Language	Derived from Perl V5.8.7
401.bzip2	93 KB	C	Compression	Lossless, block sorting data compression
403.gcc	4.3 MB	C	Compiler	gcc version 3.2
429.mcf	79 KB	C	Combinatorial Optimization	Vehicle scheduling. Uses a network simplex algorithm to schedule public transport.
445.gobmk	4.1 MB	C	AI	Plays the game of Go
456.hmmer	367 KB	C	Search Gene Sequence	Protein sequence analysis using profile hidden Markov models
458.sjeng	202 KB	C	AI	A highly-ranked chess program
462.libquantum	55 KB	C	Quantum Computing	Simulates a quantum computer, running Shor's polynomial-time factorization algorithm
464.h264ref	778 KB	C	Video Compression	A reference implementation of H.264/AVC
471.omnetpp	870 KB	C++	Discrete Event Simulation	Discrete event simulator to model a large Ethernet campus network
473.astar	59 KB	C++	Routing	Pathfinding library for 2D maps
483.xalancbmk	6.1 MB	C++	XML Processing	Transforms XML documents to other document types
Floating point Benchmarks				
410.bwaves	50 KB	Fortran	Fluid Dynamics	Computes 3D transonic transient laminar viscous flow
433.milc	174 KB	C	Physics	A gauge field generating program for lattice gauge theory
434.zeusmp	424 KB	Fortran	Physics	Simulation of astrophysical phenomena.
435.gromacs	1.3 MB	C,Fortran	Biochemistry	Simulate Newtonian equations of motion for hundreds to millions of particles
436.cactu-sADM	1.0 MB	C,Fortran	Physics	Solves the Einstein evolution equations using a staggered-leapfrog numerical method
437.leslie3d	185 KB	Fortran	Fluid Dynamics	Large-Eddy Simulations with Linear-Eddy Model in 3D.
444.namd	342 KB	C++	Biology	Simulates large biomolecular systems.
447.dealII	4.1 MB	C++	Finite Element Analysis	Adaptive finite elements and error estimation.
450.soplex	522KB	C++	Linear Programming	Solves a linear program using a simplex algorithm and sparse linear algebra.
453.povray	1.5 MB	C++	Image	Image Ray-tracing
454.calculix	2.0 MB	C,Fortran	Structural Mechanics	Finite element code for linear and nonlinear 3D structural applications.
459.gemsFDTD	596 KB	Fortran	Electromagnet	Solves the Maxwell equations in 3D
465.tonto	4.9 MB	Fortran	Chemistry	quantum chemistry package
470.lbm	22 KB	C	Fluid Dynamics	Lattice-Boltzmann Method to simulate incompressible fluids in 3D
482.sphinx	234 KB	C	Speech recognition	A widely-known speech recognition system

Table 4.1: Description of SPEC CPU2006 benchmark executables

	Loop Nest Coverage				Loop Nest Coverage		
Benchmark	All	⩾ 1%	⩾ 10%	Benchmark	All	⩾ 1%	⩾ 10%
400.perlbench	1059	33	25	401.bzip2	211	33	12
403.gcc	6273	23	3	429.mcf	78	14	5
445.gobmk	999	101	27	456.hmmer	943	10	8
458.sjeng	260	71	44	462.libquantum	111	15	11
464.h264ref	1405	28	25	471.omnetpp	678	16	4
473.astar	122	33	16	483.xalancbmk	6785	48	2
410.bwaves	84	18	10	433.milc	398	56	28
434.zeusmp	599	96	4	435.gromacs	2287	12	4
436.cactusADM	1382	4	2	437.leslie3d	383	48	4
444.namd	617	46	10	447.dealII	10455	23	5
450.soplex	1020	15	8	453.povray	1927	20	2
454.calculix	4090	12	4	459.GemsFDTD	1021	32	9
465.tonto	9669	12	4	470.lbm	34	2	2
482.sphinx3	696	42	5	Mean	1996	37	13

Table 4.2: Recognised Loop count after no filter, 1% filter and 10% coverage filter for SPEC2006 benchmark binaries compiled by gcc with O3 optimisations.

overall coverage for each loop is calculated as the maximum of the individual coverage collected from each set of training input. Table 4.2 gives the number of loops that are above 1% and 10% coverage threshold which is measured over all available training inputs. From the coverage distribution, we observe that, on average, 2% of statically recognised loops have coverage over 1% and only 0.6% of the loops have over 10% coverage. Most of the static loops have low coverage or are not even executed for training inputs.

With the coverage information of each loop, loops with low coverage are filtered out. Low-coverage loops are not subject to further parallelism analysis as they never deliver great overall program performance even with a high degree of parallelism. For example, loops with less than 1% coverage can bring a maximum limit of 1.1% of whole performance improvement according to Amdahl's Law. The cost of thread control and synchronisation would easily negate the tiny performance gain. The filtering significantly reduces the amount of analysis work for parallelism analysis.

4.1.4 Dynamic Data Dependence Profiling

With significantly fewer loops, it is much faster to perform much more heavy-weight profiling specifically on the loops with high coverage. For each loop of interest, it is passed to BEEP for dynamic data dependence profiling. The BEEP static binary analyser generates hint programs that annotate all register, memory and special accesses such as induction variable accesses. Boundaries of loop iterations are also annotated. During instrumentation, the BEEP binary emulator interprets the hint program and feeds read, write accesses and boundary events to parallel execution models. The result of the data dependence profiling from BEEP is typically a dynamic dependence graph (DDG). There are considerably many different approaches for parallelisation, but all have to respect the data flow nature of the program. Each parallel execution model is essentially scheduling the nodes of the data dependence graph to different execution units, while maintaining correctness through respecting all dependencies specified by the original data flow order.

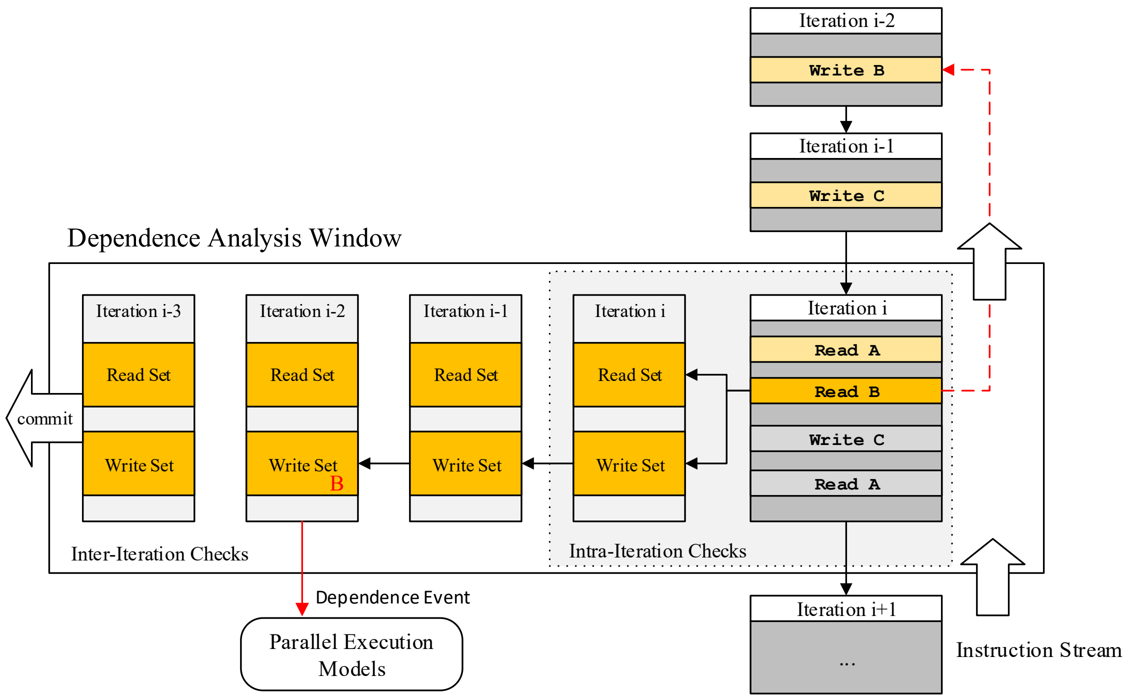


Figure 4.2: Dynamic dependence analysis using windowing

There is numerous work [114] that analyses execution traces to build a DDG to analyse parallelism. The trace-based technique uses too much computation and memory footprint, and is not scalable to analyse big applications with long execution time. For example, the memory trace for SPEC2006 464.h264ref for training inputs is as large as ten GB. Despite using compression schemes such as SD3 [115] to reduce memory usage, the scalability for programs with a large number of accesses are not resolved.

To quickly investigate parallelism with limited resource requirement, I propose a window-based dependence analysis structure. A window consists of a fixed number of hypothetical execution units which are assumed to execute in parallel. Each unit is assigned to execute a sequence of dynamic instructions called *epochs*. An epoch has the same version of dynamic instances from their original static instructions. Normally for loop level parallelism, an epoch is typically interpreted as an instance of a loop iteration. Using epochs to evaluate parallelism was firstly discussed by Steffen and Mowry [116] and the model was also used in Niall Murphy's work [112].

The window (Figure 4.2) is a first-in-first-out (FIFO) runtime buffer which maintains a history of read and write accesses of the last fixed N number of epochs. Whenever a new iteration is started, the window shifts along by freeing the oldest epoch and allocating spaces for the new iteration. The windowing approach is the analogue of N hardware threads taking turns to execute epochs in a round robin order. From the data-flow point of view, the effect of a window is to unroll a sophisticated data dependence graph and navigate from the top N instances of the static graph to the last N instances of the graph.

Detecting data dependencies in a window would no longer be able to find cross-iteration dependence that goes beyond the number of epochs N. To address the dependence distances beyond N, the parallel execution models are refined with extra assumptions: the execution units have to commit their changes in order. It means an execution unit finishes iteration i and continues to execute iteration $i+N$. So that a write in iteration i is guaranteed to occur before the writes in iteration $i+N$. As threads commit in serial, dependence distances beyond N would be positioned in main memory as a read-only value. The assumption is also in coordination with the constraints from real hardware with fixed number of cores.

As discussed in Section 2.2.2, three major types of dependencies (RAW, WAR, WAW) are checked and recorded. For cross-iteration dependencies, the first read of an address would be checked against all the write sets from epochs in the window from the reverse order that execute previous iterations (shown in Figure 4.2). In addition to building the data dependence graph, whenever a data dependence is discovered, it generates an event that would invoke callbacks of various parallel execution models. For example, Figure 4.1 (right) shows that a synchronisation parallel execution model may invoke delay costs in thread communication, and a speculation model may simulate costs for transaction abort and re-execution in the same event.

With the help of windowing, dynamic read/write access streams can be analysed on-the-fly as we execute the application. A temporary dependence graph of a loop could be easily constructed and destructed as controlled by hint programs during any phase of the application. The storage footprint is therefore significantly less compared to trace-based or compression-based approaches.

4.2 Ideal Parallel Execution Models

One could wonder what the ultimate upper limit of parallelism is for a given loop, assuming the loop is executed by an ideal parallel machine. Theoretically, it is well explained by Amdahl's law:

$$Speedup = \frac{1}{(1-p)+\frac{p}{n}} \tag{4.1}$$

where p is the parallel fraction and n is the number of threads provided by the parallel machine. However, Amdahl's law assumes the parallel fraction p to be uniformly executed by n threads. For general applications, those embarrassingly parallel regions are rarely seen. Instead, irregular patterns of control and data patterns are more common.

There are many ways of scheduling the work of a loop for parallel execution, and each division would result in a different performance. To enable a fair comparison, we normalise each irregular and implicit parallel fraction p' from the given parallel paradigms. The normalised parallel fraction p' generates the same speedup as if it were executed in an embarrassingly parallel region. Regardless of the parallel paradigms used, *Speedup* is the indicator of the parallel fraction of the program execution that could be overlapped. In this section, three ideal parallel execution models are evaluated.

4.2.1 DOACROSS Dataflow Model

The simplest and scalable parallel model is the ideal DOACROSS Dataflow Model. As discussed in Section 2.3.2, DOACROSS parallelism requires no code transformations and allows different loop iterations to be scheduled to threads in a round-robin order. Dataflow edges beyond iteration boundaries are resolved through forwarding written values to the next thread using synchronisation. This model can also recognise DOALL loops once it finds no value was forwarded except induction variables.

The fundamental ideal assumption is that writes of the current thread are immediately available to the next thread, which is similar to an ideal parallel machine that has zero inter-thread communication overhead. It also assumes all writes in the current thread are perfectly privatised, buffered and accessed internally with zero cost. Internal writes do not interfere with early or later writes to the same address from other threads. The assumption can remove the effects of WAR and WAW dependencies and leave true dependencies for evaluating the true parallelism. The model analyses traces of dynamic instruction streams that access registers and memories. No control flow is considered; Intra-iteration control dependencies are also ignored in this ideal model.

For speedup evaluation, the parallel execution model initialises an array of cycle counters for N execution units. Each instruction is counted as one cycle to execute and the corresponding cycle counter is incremented based on the scheduling policy. When there is a true cross-iteration dependence, the current thread reveals the conflicting write with a "time stamp". The depending read from the next thread must not occur earlier than the write. Therefore cross-iteration dependencies are enforced by delaying the time stamps from depending reads to wait for availability of output writes from the previous epoch. The delay, as a consequence of dataflow constraints, prevents threads from completing its task independently. The delay effect is demonstrated in Figure 4.5, where red dashed edges represent cross-iteration dependencies. The final speedup of the loop is determined by the largest timestamp from all cycle counters over the sequential cycle count for the loop.

We selected all recognised (around 800) static loops with over 1% coverage filtered from the loop coverage results shown in Table 4.2. For each selected loop, we use BEEP to evaluate ideal parallelism under the DOACROSS Dataflow model. The estimated ideal speedup for each loop is then calculated. Figure 4.3 shows the scatter plot of the calculated ideal speedup and the loop coverage. Overall the graph shows an interesting phenomenon, a significant proportion of the loops reside in the bottom-left region with low coverage and low speedups. These are the loops with low parallelism due to high serialisation specified from their code. For loops in the other two corners: high-coverage with low speedup (bottom right) and low-coverage with high speedup (upper-left), still fail to deliver effective overall application performance. Loops towards the upper right corner represent good candidates for parallelisation.

We define *effective speedup* as the product of the estimated speedup (minus 1) for the loop and the coverage of the loop, to denote the impact on the overall performance gain if the loop is parallelised. Slowdown (speedup less than 1) can be evaluated as a negative effective value.

$$\texttt{effective\_speedup} = (\texttt{speedup} - 1) \times \texttt{coverage} \tag{4.2}$$

Loops below a certain effective speedup threshold are not suitable for parallelisation, which is shown as the red line in Figure 4.3. The value of the threshold could be calibrated for different hardware. Loop points above the threshold line can be selected for parallelisation. Suppose for example an ideal speedup of 1.5× can be achieved on a loop with 10% coverage as the threshold. The threshold value is $(1.5 - 1) * 0.1 = 0.05$. It is found that even with an ideal effective speedup threshold of 0.05, there are only around 3% of loops that could increase overall program performance.

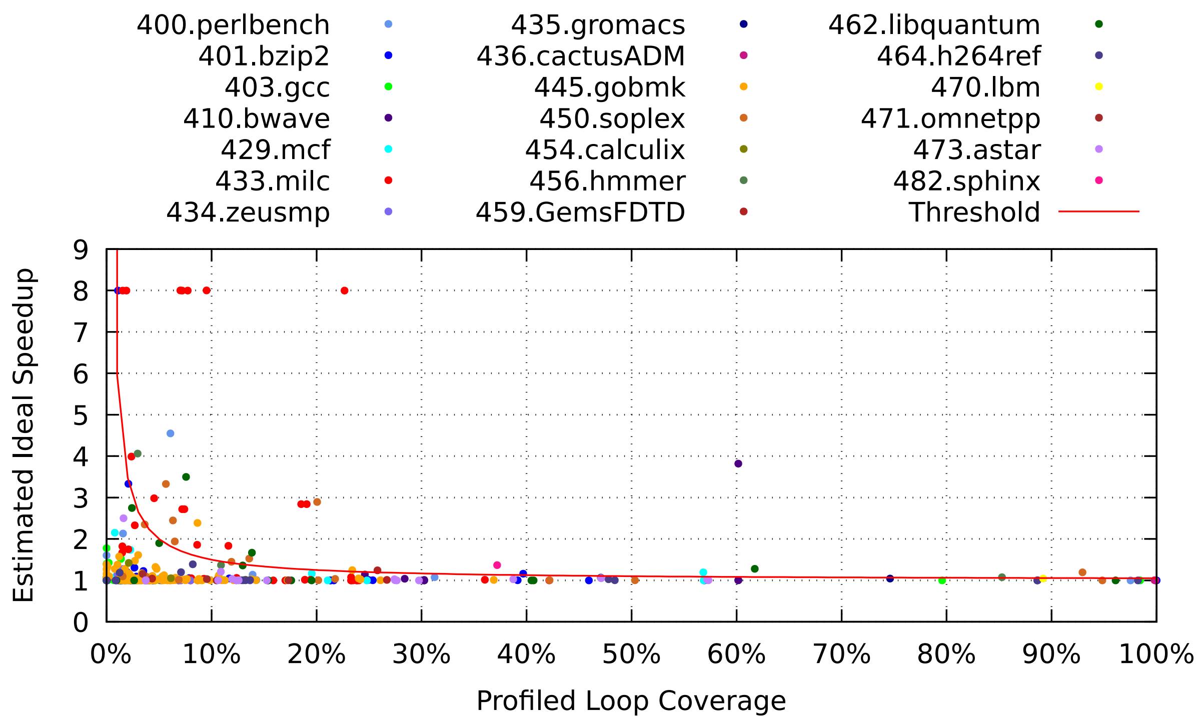


Figure 4.3: Estimated speedup for all big SPEC2006 loops on a hypothetical 8-core machine with zero communication overhead. Each point represents a static loop candidate with over 1% coverage from profiling training inputs. Dots with the same colour represent loops from the same benchmark. The red line represents an ideal effective speedup threshold of 0.05

4.2.2 Induction/Reduction Optimisation

The previous dataflow model assumes zero communication overhead, which reflects an upper bound of the parallelism while respecting all data dependencies specified in the executable. It is found most binaries don't exhibit enough parallelism even in ideal cases. However, it is possible that even more parallelism could be exposed by further breaking data dependencies using induction and reduction optimisations, a subset of value prediction optimisation. Instead of waiting for the previous thread to forward value in the dataflow model, the current thread can predict and calculate the value, which effectively removes the data dependence pair and relaxes the constraints.

It is certain that all loops contain at least one variable that defines the control flow of the loop. For most simple loops, the controlling variable is typically an induction variable. An induction variable is a variable that gets increased or decreased by a fixed amount on every iteration of a loop. Therefore it is relatively easy to calculate the corresponding value of the induction variable and its generated values for other depending variables for a thread based on the iteration number it is assigned. Therefore all delays and subsequent calculations from waiting or updating induction variable could be removed. Other forms of variables, such as variables that control linked list traversal, could not be easily generalised. But they can be predicted correctly with a dedicated software routine to re-calculate the depending variable for each iteration. If the software routine could not be summarised in a closed form, we can perform prediction of the value with an extra cost of validation and buffering. The approach is similar to *thread-level speculation*, which is

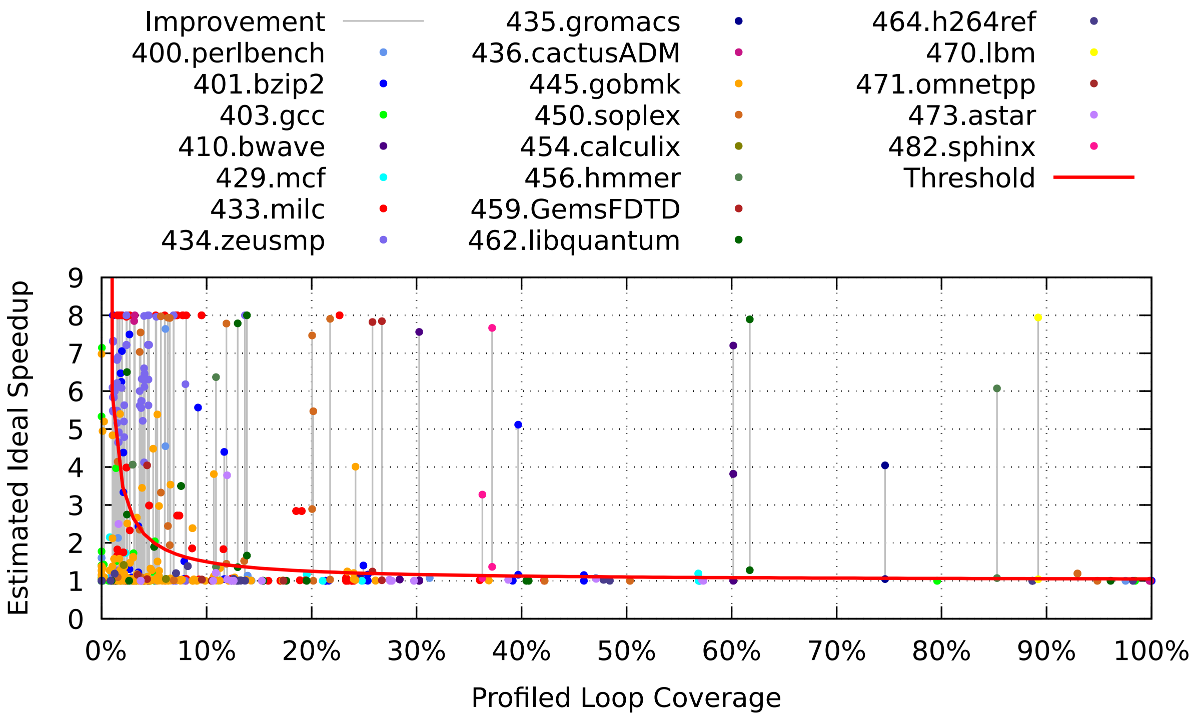


Figure 4.4: Estimated speedup improvement for all big SPEC2006 loops on a hypothetical 8-core machine with induction and reduction variable optimisation compared to the dataflow execution model in Figure 4.3.

discussed in the speculation model in Section 4.3.

There are other common patterns of data dependencies such as reduction operations. Reductions are calculations which accumulate multiple values to a single value. Common examples of reduction operations include summing the contents of an array, finding the maximum value in an array and counting the number of elements of a particular type in an array. All of these operations are implemented with a single variable to hold the reduction value. The value typically forms a cross-iteration dependency. Any computation of this sort in which the reducing operation is both commutative and associative. The order of accumulations can be relaxed and it can be divided into thread-private sub-reductions and a final combining reduction once the loop completes.

The induction/reduction model evaluates the extra parallelism that induction and reduction variables can improve on top of the ideal dataflow assumptions. However, it is not easy to conclude induction and reduction operations purely through dynamic binary instrumentation. Therefore, the model relies on BEEP static binary analysis to recognise all induction and reduction variables for each loop. Through static binary analysis, all instructions that access and modify induction/reduction variables are recorded and tagged to the corresponding basic block. The information is encoded into static hints and the binary emulator in BEEP would read the hints whenever it executes the corresponding basic block. Once it tracks the exact location of the induction/reduction variable, it generates an event to the parallel execution model on induction/reduction variables. The operation for the induction/reduction variable in the model is simply ignoring the delay caused by the induction/reduction variable. All subsequent reads and writes to the variable inherit an earlier timestamp due to the elision of the delay.

The limitation of the model is that it is highly sensitive to the accuracy of the static binary analysis. If induction/reduction variables can not be easily recognised statically due to lack of symbolic information, the model would see no optimisation opportunities compared to the original dataflow model.

Figure 4.4 illustrates the performance improvement on top of the original dataflow model with the same group of loops to be re-evaluated under the induction/reduction model. From the graph, we can see that a fraction of the loops saw a substantial performance boost by relaxing the constraints of induction and reduction variables. Therefore, the induction and reduction optimisation is an essential procedure to expose parallelism from binaries. All the loops that have full speedup (dots along the horizontal line around 8) represent DOALL loops, which have no other cross-iteration dependencies besides induction and reduction operations.

4.2.3 Code Motion Model

The estimated speedup from the DOACROSS dataflow model and induction/reduction optimisation are calculated based on the delay caused by enforcing cross-iteration dependencies that are not easily removed. The delay is calculated by the timestamp difference when the sender thread's write value is available and the timestamp when the receiver thread's read is requested. However, we demonstrate in Figure 4.6 that the delay can be shrunk by scheduling independent instructions with ready operands. In this way, the sender thread can generate its write much quicker to hide the delay of the data dependence. The dataflow model relies on the original read/write order specified from the input binary. However, it was generated by compilers as specified in source code, which is not intended to expose parallelism.

Altering the order of instructions would help to expose more parallelism from the binary. Figure 4.6 shows the delay hiding effect by instruction reordering from the example loop in Figure 2.9. The code motion model is implemented to reorder instructions and recalculate the delay based on the same assumptions of dataflow and induction models. *Code Motion* is a term typically used by the *loop invariant* compiler optimisation that moves invariant code outside the loop. In the concept of DOACROSS parallelisation, *Code Motion* refers to moving instructions within a loop while respecting the original data dependencies to minimise the distances of cross-iteration dependencies.

Determining the optimal instruction order to minimise the delay from cross-iteration dependencies is the key to the code motion model. Calculating the most optimal order is difficult, both compute and memory intensive. Dynamically, it has to perform sorting and scheduling on the whole dynamic instruction trace of the loop. It is impractical for loops with millions of iterations due to its poor scalability. However, if instructions from each loop iteration are repeated in the same pattern, the problem can be simplified as reordering the static instructions from the loop iteration. For a loop with multiple control flow paths in its iteration, determining the optimal order statically is a NP-complete problem. Instead, we seek a sub-optimal solution, where it alters its dynamic order so that the forward delay to the next dynamic iteration is minimised. While it might not be the best solution, as it only minimises the delay of the next iteration instead of all instances, the assumption is sufficiently ideal to calculate the ultimate upper bound of parallelism.

The implementation of the code motion model has to record all the cross-iteration

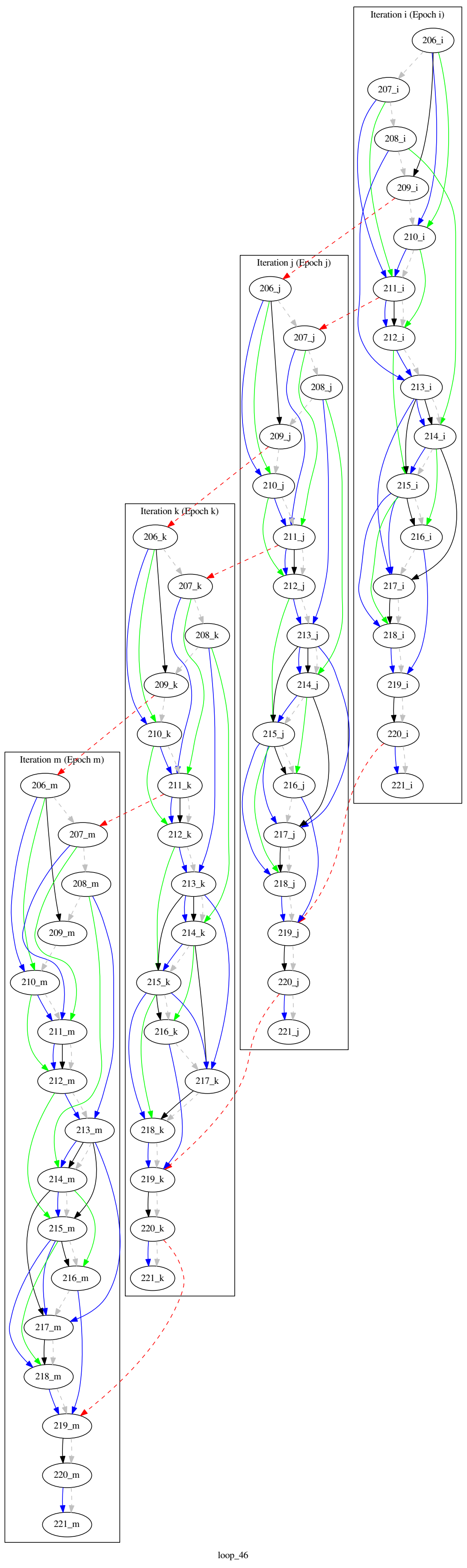


Figure 4.5: Unrolled DDG for the example from Figure 2.9 from BEEP: original code order

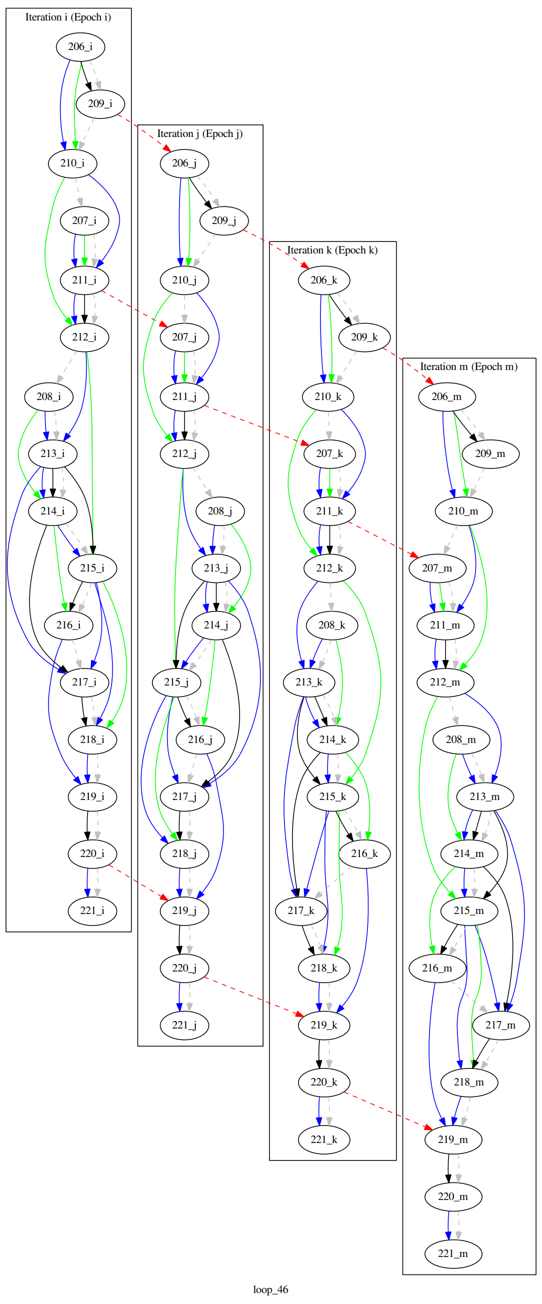


Figure 4.6: Unrolled DDG for the example from Figure 2.9 from BEEP: optimised code order

dependencies in the dependence analysis window (Figure 4.2). It marks the *inputs* and *outputs* of intra-iteration DDGs in each epoch. As shown in Figure 4.6, the origin and destination of cross-iteration edges (red dotted arrow) is labelled as inputs and outputs respectively. The information for all the output nodes is only fully collected when data dependencies from subsequent N iterations in the window have been computed.

Algorithm 1 Rescheduling for Dynamic Optimal Order

```
Require: ddg = $G(\mathbf{N}, \mathbf{E})$                    ▷ DDG must be a DAG, no cycles allowed
Require: out                                   ▷ Set of output node
Ensure: $\forall n \in \mathbf{N}, pri(n) =$ MAXINT              ▷ Initially all priority is max int
 1: procedure SchedPriority($n$)   ▷ Return the minimum steps to an output node
 2:     if $n \in$ out then
 3:         pri($n$) = 0
 4:     end if
 5:     for all $p \in pred$(out) do
 6:         pri($n$) = min(SchedPriority(p)) + 1
 7:     end for
 8:     return pri($n$)
 9: end procedure
10: procedure Reorder
11:     for all $n \in \mathbf{N}$ do
12:         pri($n$) ← SchedPriority(n)              ▷ Prepare priority of all nodes
13:         weight($n$) ← size(pred($n$))     ▷ Weight is the number of predecessors
14:     end for
15:     for all $n \in \mathbf{N}$ do   ▷ Insert all nodes with no predecessors into priority queue
16:         if weight($n$) = 0 then
17:             priQueue(pri($n$)).insert($n$)
18:         end if
19:     end for
20:     while !priQueue.empty() do
21:         $n$ ← priQueue.pop()
22:         order.push($n$)           ▷ Pop from priority queue and output to the order
23:         for all $s \in succ(n)$ do
24:             weight($s$) ← weight($s$)−1
25:             if weight($s$) = 0 then
26:                 priQueue(pri($s$)).insert($s$)
27:             end if
28:         end for
29:     end while
30: end procedure
```

An optimal DOACROSS order for a given epoch is an instruction schedule to *schedule the output nodes as early as possible* and respect the original data dependencies. As shown in Algorithm 1, each node from the data dependence graph is marked with two properties, the weight and priority. The priority is defined as the minimum steps from the current node to the nearest output node. The closer to the output node, the higher priority that it should be scheduled earlier. The weight property represents the number of predecessors

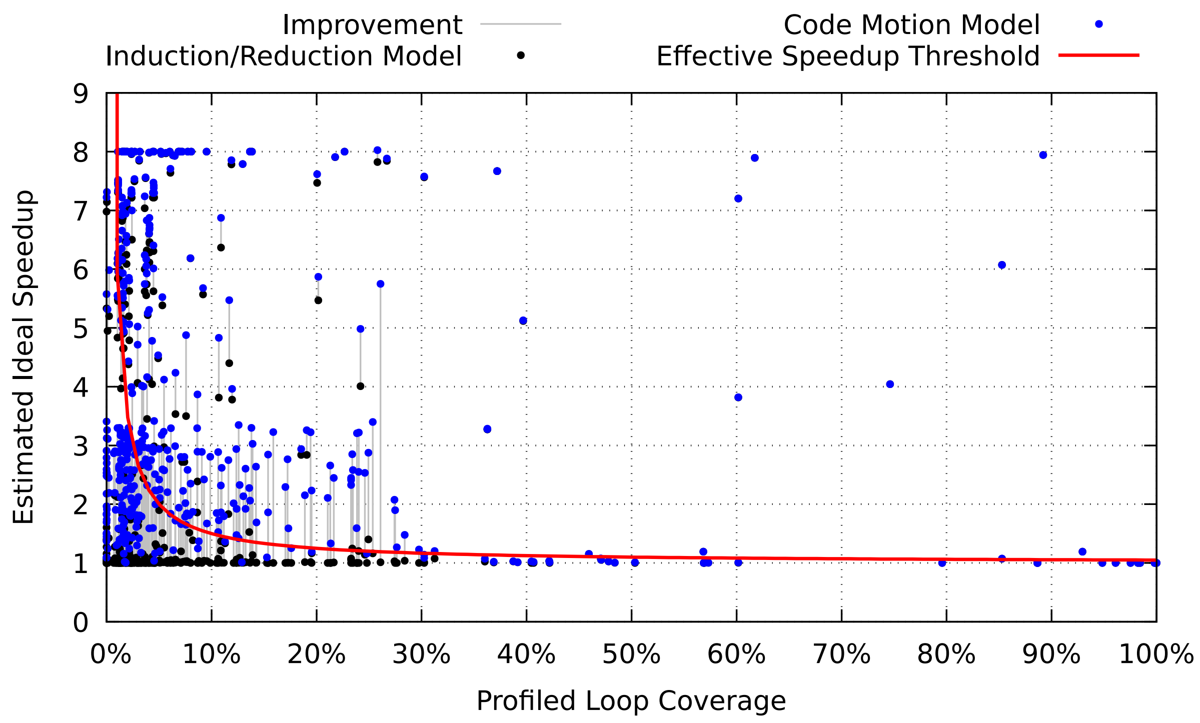


Figure 4.7: Estimated speedup improvement for all big SPEC2006 loops on a hypothetical 8-core machine with code motion optimisation compared to the induction/reduction execution model in Figure 4.4. Estimation for loops with large coverage is limited due to scalability problems.

that the node should wait. When a node is scheduled, the weight of all its successors decreases by one to denote the subsequent operands that are referring to this node are available. Once the weight is decreased to zero, it means all its sources are ready and the node can be scheduled. If there were multiple nodes with all sources ready, the node with the highest priority (low value) is scheduled first. A priority queue is used for sorting the nodes based on the priority and dequeuing one node at a time with the highest priority. Note that the algorithm is non-deterministic, since for nodes with the same priority, it would randomly select one node and push it into the order. Therefore there exist more than one solution for the output instruction order.

When finding the least path to an output node, there should not be any cycle in the graph. Therefore the algorithm requires the DDG for each graph to be directly acyclic. For each DDG constructed in the epoch, strongly connected components (SCCs) must be recognised, and the DDG must be coalesced based on the SCCs. When the reordering is applied to the coalesced DDG, the whole SCC is reordered as a single node for reordering. The drawback of this algorithm is that it does not consider the priority difference of the output node. As each output node is responsible for causing delays for a specific set of iterations, it is challenging to quantify the differences of the depending iterations. Moreover, the algorithm has the complexity of $O(n^3)$ as it uses a priority queue for priority-based sorting for all n nodes. It requires significant computing resources when instrumenting complicated executables with millions of memory accesses and what's worse, it performs reordering for each single iteration.

Figure 4.7 shows the further improvement that is brought by the code motion model

Benchmark	Loop Count Coverage⩾ 1%	Loop Count Effective	Benchmark	Loop Count Coverage⩾ 1%	Loop Count Effective
Integer Benchmarks					
400.perlbench	33	1	401.bzip2	33	10
403.gcc	23	0	429.mcf	14	4
445.gobmk	101	8	456.hmmer	10	6
458.sjeng	71	0	462.libquantum	15	5
464.h264ref	28	6	473.astar	33	4
Floating Benchmarks					
410.bwaves	18	2	433.milc	56	15
434.zeusmp	96	48	435.gromacs	12	3
436.cactusADM	4	1	437.leslie3d	48	2
444.namd	46	1	447.dealII	23	2
450.soplex	15	10	453.povray	20	3
454.calculix	12	4	459.GemsFDTD	32	4
465.tonto	12	0	470.lbm	34	4
482.sphinx3	42	2			
Total	831	145			

Table 4.3: The fraction of loops that are estimated to achieve effective speedup from ideal execution models from SPEC2006 benchmarks. The effective speedup threshold is set to 0.05

on top of the induction/reduction model. More loops are lifted to above effective speedup threshold and demonstrate higher ideal speedup. There are loops that hardly see improvement because most of their loop body is treated as a single or multiple big SCCs. There is not too much space left for reordering big SCCs. For some big loops, the code motion model fails to run from SPEC2006 executables. Due to the scalability problem, some loops with high coverage may takes days to simulate. The scalability problem of the model is not the focus of this dissertation. It could be addressed by removing invariant nodes and reducing the node count of the big loop in the static analysis.

Despite the absence of loops with high coverage, we see there is significant parallelism in loops in SPEC2006 benchmarks proved from ideal parallel models. The final speedup reflects the ultimate upper bound of the DOACROSS loop parallelism. If a loop is estimated with a low speedup even with ideal assumptions, it would never be a good candidate for parallelisation regardless of any further realistic estimation. Granted we could further exploit more parallelism if we break more data dependencies with the help of understanding the original algorithm and data structures. With the current implementation of static analysis, it is difficult to obtain enough information to further remove data dependencies and release more parallelism. More comprehensive static binary analysis is beyond the scope of this dissertation. Table 4.3 shows the number of filtered loops that prove effective speedup over whole program execution. These filtered loops are selected for further realistic investigation.

4.3 Realistic Parallel Execution Model

So far we investigated the ideal parallelism from SPEC2006 binaries. This section evaluates filtered loops with more constraints under realistic considerations in both static

Producer

```
SIGNAL(c):
    while (c.consumed==0);
    write(c.data);
    c.consumed = 0;
    c.ready = 1;
```

Consumer

```
WAIT(c):
    while (c.ready==0);
    read(c.data);
    c.ready = 0;
    c.consumed = 1;
```

Figure 4.8: Pseudo code for the signal and wait operations in thread synchronisation for a given channel (assuming TSO and atomic load and store)

analysis and dynamic operations. For static analysis, due to the lack of symbolic information and ambiguity caused by inputs and irregular control flows, not all data dependencies could be accurately identified by the static analysis. To maintain correctness, these undecided memory accesses are also treated as data dependencies even if they are not necessary. As a conservative consequence, it requires additional runtime handling that causes unnecessary runtime overhead.

The second group of realistic assumptions is adding the costs of performing parallel operations and synchronisations in real parallel hardware. The realistic model assumes that parallel execution is performed in a typical multi-core architecture with no special hardware extensions to accelerate parallel execution (e.g. no register forwarding between cores, no hardware transactional memory, etc.). The majority of multi-core systems are designed according to the von Neumann architecture, where inter-core communications are performed through memory or cache hierarchy. By forwarding values, a thread has to write the value to a memory location, and the other thread can load from it to retrieve the value. The cycle cost that data is traversed from one core to the other through cache hierarchies and memory systems can be calibrated from real systems.

Although it is more accurate to simulate a full cache and memory hierarchy during binary instrumentation, it is typically time-consuming and scales badly. Moreover, it is impossible to accurately simulate the cache behaviour under a real multi-threaded environment. Therefore for simplicity, the realistic model only assumes a fixed constant cycle cost for forwarding data between threads.

4.3.1 Synchronisation Model

As discussed, the realistic constraints can be summarised as ambiguities in static analysis and cycle costs in thread communication. The synchronisation model is implemented to estimate the impact of these two realistic constraints. For static constraints, all static-undecided memory accesses are marked with `STATIC_UNDECIDED` hint instructions. During instrumentation and evaluation in BEEP, these dynamic accesses would result in extra cycle penalties in the model. The penalty delay is calculated by adding an extra data dependence from the previous write to the current read location. For comparison, the model also evaluates the synchronisation cost if we assume all data dependencies are fully accurately recognised in the static analysis.

The model assumes a shared memory model without hardware for accelerating synchronisation. A software handshake scheme is required to ensure coherence and correctness. The correctness of sequential execution requires that threads must be coordinated to send and receive data deterministically. An extra memory location (ready flag) is allocated to denote the validity of the data. On the one hand, the receiver/consumer must wait until the ready flag is set, and it must unset the flag after it consumes the data, the operation is called **`WAIT`** operation. On the other hand, the sender/producer also has to wait until the flag is unset since the receiver has not consumed previous data. The sender must also set the flag after it sends the data, which is refereed as **`SIGNAL`** operation. Both operations are in critical sections and have to be guarded in locks, which are shown in Figure 4.8. The `WAIT` and `SIGNAL` operations create a chain of sequential execution across the cores which ensures that any code involved in loop-carried dependencies executes in loop iteration order. We define the *synchronisation cost* to be the cycle cost that threads are spent to perform *signal, propagation and wait* when forwarding values.

The model maintains N virtual cores and a cycle counter for each core. Each virtual core is assigned with a loop iteration in the round-robin order of the DOACROSS style. It labels each register/memory access a timestamp according to its cycle counters from the dynamic iteration. Runtime conflicts are checked based on timestamps on-the-fly as the application executes. To resolve a conflict, the model calculates the delay of each core based on the following principles:

- A thread propagates data only to its adjacent thread, even if the next thread doesn't need the data for its iteration. This is to reduce the complexity of thread communication and prevent potential data hazard for long dependence distances.

- If there is a cross-iteration dependence, the time stamp for the last write by thread 1 must be no earlier than the time stamp of the last read from thread 2 plus the propagation cost. The write of thread 1 must wait for thread 2 to consume its previous read values in order to update the new write value. It is to simulate the SIGNAL lock shown in Figure 4.8.

- If there is a cross-iteration dependence, the time stamp for the first read of thread 2 must be no earlier than the last write of thread 1 plus the propagation cost. This is to simulate the process of the WAIT to ensure the data is propagated correctly.

There are other data dependencies that only occur along a specific control path of an iteration, it is not always the case that a signal is invoked at every iteration. Improper handshakes would cause deadlock on the wait of the next thread when there are no signals propagated. Therefore, signal and wait operations must be inserted at every iteration, even if the data is not required for this instance of iteration. This operation creates additional overheads in synchronisation, due to this thread control mechanisms. The model adds this extra cost when a dynamic data dependence pair is not occurring in a thread, but it still invokes a signal cost for the thread.

4.3.1.1 Ambiguous Static Binary Analysis

To demonstrate the penalty caused by the ambiguity of static analysis, I select one loop candidate from each of the SPEC2006 benchmarks that demonstrates effective speedup through ideal parallel models. The selected loops are evaluated through the synchronisation model under 8 hypothetical threads.

Firstly, a light-weight static alias analysis is implemented to detect potential data dependencies. The detailed alias analysis implementation is discussed in Section 5.2.2.

Figure 4.9: Parameters used for the synchronisation model

Parameters	Estimated Cycle Cost
Average communication cost	50 cycles
Schedule threads from thread pool	100 cycles
Induction/Reduction variable privatisation cost	10 cycles

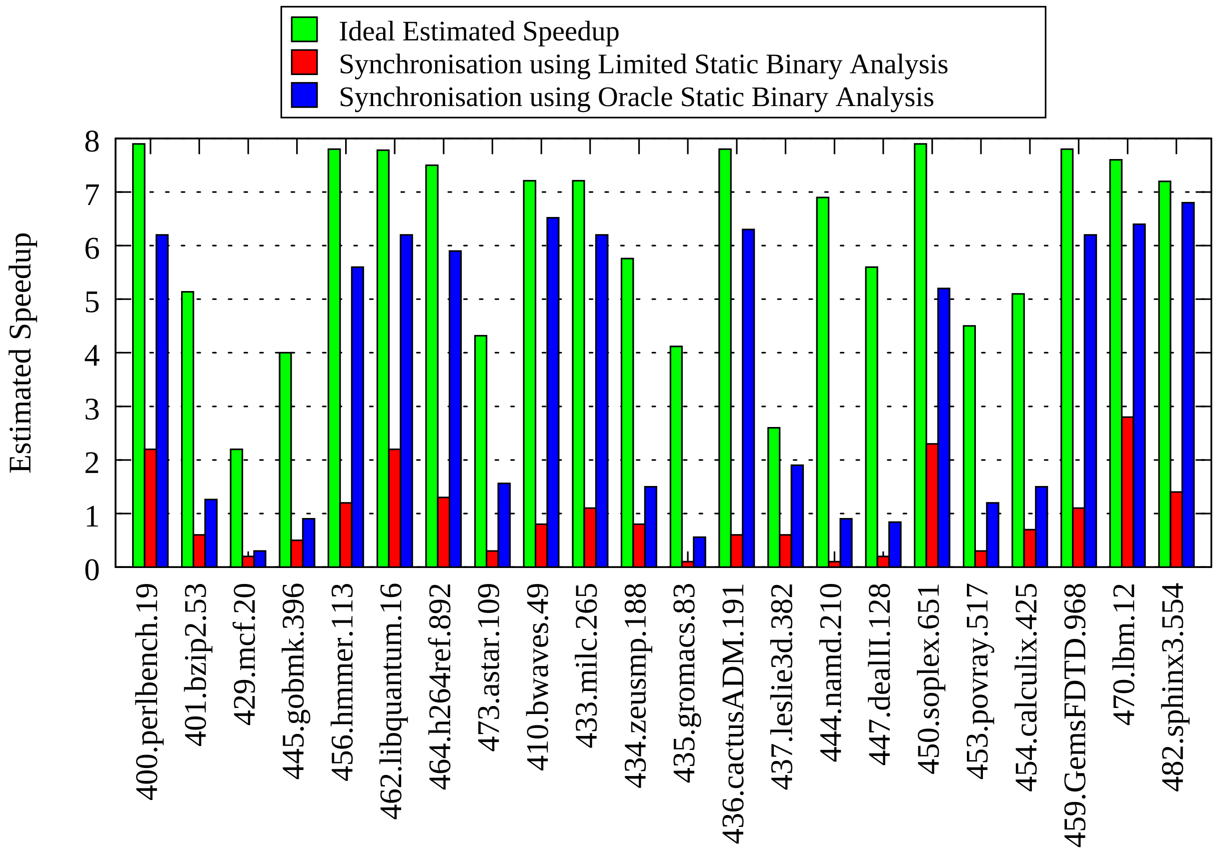


Figure 4.10: Estimated speedup of selected loops from SPEC2006 binaries of 8 threads assuming limited static binary analysis and fully accurate Oracle static analysis in the synchronisation model.

To make BEEP aware of the statically undecided memory accesses, they are annotated with a hint instruction called `STATIC_UNDECIDED` to be resolved by synchronisation. As a result, when BEEP's dynamic instrumentation finds a memory access is labelled by the hint instruction, it calculates the cost of the unnecessary synchronisation and adds the delay to the corresponding cycle counter. For comparison, a semi-ideal case assuming fully accurate static analysis is also considered. It can be implemented by ignoring the `STATIC_UNDECIDED` hint instruction.

The synchronisation model is configured with the parameters shown in Figure 4.9. It also assumes threads are created in advance and placed in a thread pool, so that the cost for thread scheduling and initialisation is relatively low at 100 cycles. All the induction and simple reduction variables are evaluated with privatisation optimisation, where only a small calculation cost is added per variable.

Figure 4.10 shows the estimated performance from the synchronisation model on 8 hy-

Synchronisation Route	Estimated Cycle Cost
Shared L2 Cache Hit	10 cycles
Shared L3 Cache Bank Hit exclusive	40 cycles
Shared L3 Cache exclusive in other bank	75 cycles
Shared DRAM	200 cycles

Figure 4.11: Estimated data propagation cost between cores

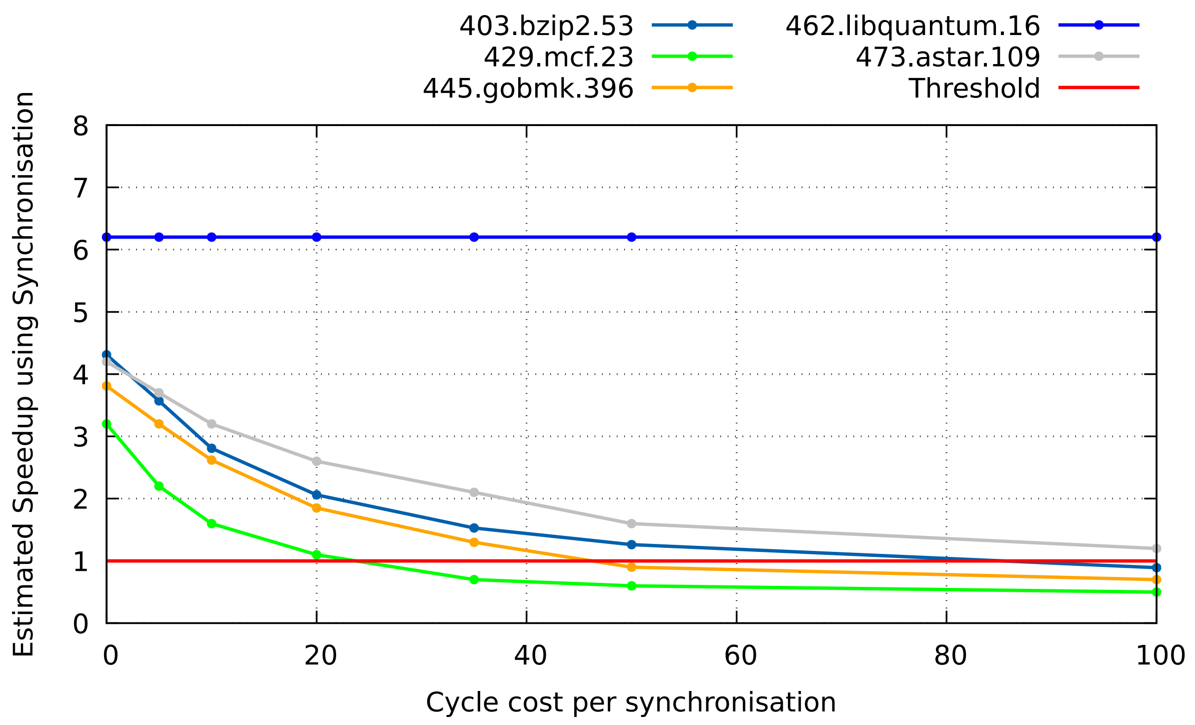


Figure 4.12: Estimated speedup of selected loops from SPEC2006 binaries of 8 threads by varying propagation costs for synchronisation in the synchronisation model. It assumes a fully accurate static dependence analysis.

pothetical cores. The loops are selected based on the filtered loops from the ideal models. Red bars represent the estimated performance if a light-weight static alias binary analysis is used to drive synchronisation. Notably, there is a significant performance degradation using the limited static analysis compared to oracle static analysis. The undecided memory accesses from the limited static analysis result in unnecessary synchronisation operations and overhead that negates final performance. Therefore the performance of the synchronisation-based parallelisation is sensitive to the accuracy of static dependence analysis.

The conclusion is different from the profiling studies by Murphy [64], where he claimed extra static analysis is not able to bring further performance gain for parallelisation. It is different because his conclusion is drawn from mature compiler alias analysis from source level. However, at binary level, the current implementation of static binary alias analysis could not reach the accuracy level of source analysis. There remains a huge potential to improve the accuracy of static binary analysis.

4.3.1.2 Thread Communication Latency

The second realistic constraint in the synchronisation model is the cost to forward data values between actual cores. Table 4.11 listed the typical propagation cycle cost to communicate between cores in a conventional machine. The propagation latency depends on the location and routes that data propagates through the cache hierarchy. To model the impact of communication on performance, we vary the latency as a parameter to be evaluated in the synchronisation model. Figure 4.12 illustrates the performance degradation of five selected loops as the propagation latency increases from 0 cycles to 100 cycles. It shows that the performance of each loop has different degradation sensitivity to the propagation delay. The sensitivity depends on the frequency and number of occurrences of its cross-iteration dependencies. For DOALL loops such as `462.libquantum.16`, performance is not affected since there is no actual data forwarding across iterations.

For other selected loops in Figure 4.12, the benefits of parallelism are negated when the latency exceeds around 60 cycles, which is below the estimated cost for synchronisation through the L3 cache. Therefore, if the data is communicated through L3 between threads, it is likely that the synchronisation cost outweighs the benefits from parallelisation. It is possible to alleviate the cost by prefetching the signal data to prepare data in L2 before the wait operation. The signal prefetch was already used in the HELIX automatic parallelisation [59] by implementing a helper thread.

From the synchronisation model, we conclude that binary parallelisation using a pure synchronisation approach is insufficient to bring performance on conventional hardware for loops from SPEC2006. Firstly, an inaccurate static binary analysis incurs significant degradation due to conservative synchronisation. Secondly, even with accurate static analysis, the propagation cost is too high for current multicore system to bring actual performance for DOACROSS-based parallelisation. It may only achieve a slight speedup when its L3 cache is guaranteed to hit during data propagation. Therefore helper threads or other hardware acceleration has to be used to achieve performance. However, DOALL or DOALL-like loops are not sensitive to the communication latency, they are considered beneficial for parallelisation.

4.3.2 Thread-level Speculation Model

Instead of using synchronisation to handle data dependencies, the alternative approach is to use the thread-level speculation (TLS) as discussed in Section 2.3.5.1. The TLS approach avoids the need to perform an accurate static analysis. But it comes with a runtime cost of potential mis-speculation including transaction abort and re-execution.

To investigate whether TLS can bring actual performance on general binaries, a realistic speculation execution model is implemented. The speculation model simulates a hypothetical multi-threaded speculative environment when instrumenting the binaries in a single thread. The model maintains the same virtual cores and cycle counters as the synchronisation model but handles runtime data conflicts differently. It assumes each thread take turns to execute a loop iteration speculatively by sandboxing thread contexts and privatising its shared memory accesses in a transaction. The model firstly calculates the cost for redirecting all its memory accesses to its private read and write buffer. The redirection is typically a thread-private hash-table lookup on the original addresses. The lookup may take non-linear time due to potential cache misses, hash table conflicts or rehashing. For simplicity, the model assumes a constant average cycle cost for memory

Parameters	Default value
Speculative read and write	10 cycles per access
Privatisation on all writes	10 cycles per access
Speculative validation	3 cycles per entry
Speculative commit	3 cycles per entry
Transaction start	$64_{integer}$ or 128_{float} cycles per check-point
Transaction abort	2 cycles per entry

Figure 4.13: Parameters used for the speculation model based on the estimation of the `JITSTM` implementation discussed in Section 5.5.1.

privatisation.

As discussed in Section 2.3.5.2, value-based software transactional memory is the simplest STM implementation that can be integrated to parallelisation at binary level. Therefore for preliminary studies, the speculation model is designed to emulate the specification of the valued-based STM at word level granularity. The following summarises the runtime events used for TLS simulation, where the runtime events are generated by the BEEP binary emulator and hint instruction interpretation.

- `Transaction_start`: the model simulates a check-point operation that emulates the operation to save the whole thread context. It records the current timestamp to represent the check-pointing time.
- `Speculative_access`: the model simulates a hash table operation that records the value of the first read of each address in its read buffer and redirects its subsequent accesses to its thread-private write buffer.
- `Read_validation`: the model simulates a verification process before the write commit. It validates that all its recorded data in its read buffer has been changed by the writes from other transactions. The validation is performed by checking the timestamps of previous writes and current reads.
- `Write_commit`: if the read validation is successful, the model reveals the transaction's write buffer to shared memory. During write commit, it marks all the writes of the current transaction with effective timestamps.
- `Transaction_rollback`: if any of the read validation fails, the model firstly calculates the cost for clearing the transaction based on its existing entries in its read and write set. It then calculates the re-execution cost by finding the cycle difference between the recorded check-point timestamp and the timestamp before read validation.
- `Privatised_write`: if a memory write is statically proved that there is no cross-iteration dependence related to the write address. It is still subject to redirection for thread isolation of a transaction.

The model uses a lazy conflict detection scheme for read validation. It avoids the need to simulate validation during each speculative runtime access in eager checking. It is mainly to improve the scalability of the simulation during binary instrumentation. Simulating different conflict schemes would result in different performance estimation as

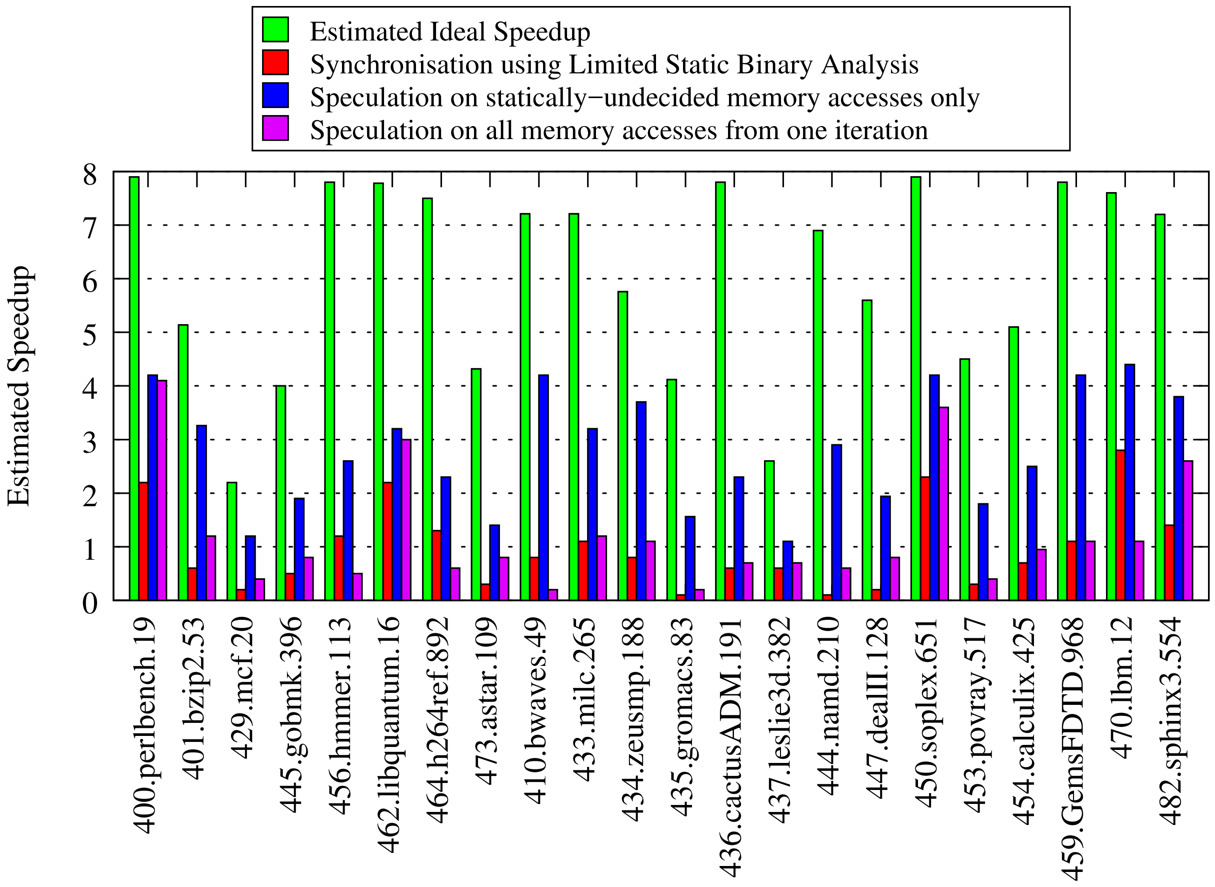


Figure 4.14: Estimated speedup of selected loops from SPEC2006 binaries on parallelisation with 8 threads using thread-level speculation

their rates of misspeculation and re-execution differ. This dissertation is only to perform feasibility investigation of the speculation technique on binaries. More accurate analysis of different design spaces of STMs are beyond the scope of this dissertation.

Figure 4.13 illustrates the basic parameters used by the speculation model in this investigation. The model casts a uniform cost of memory redirection for speculative reads and writes. It also considers per-entry costs for validation, commit and abort. Therefore it is essential to reduce the number of speculative accesses in a transaction as much as possible. As the thread-level speculation is intended for resolving the ambiguity from static binary analysis, the model only surrounds the statically undecided memory accesses into a transaction.

In the static binary analysis, if it proves there is a clear cross-iteration dependence for a memory access, it forwards the data instead of performing speculation on the address, since it is guaranteed to cause transaction aborts. The mis-speculation penalty is guaranteed to be larger than synchronisation for lazy conflict checking. If it verifies there are no cross-iteration dependencies, the cost for unnecessary speculation can be avoided. Similarly, as other models, this static proof is delivered by BEEP hint programs.

Figure 4.14 illustrates the performance estimated by the speculation model. By comparing the difference from the synchronisation result using limited static analysis shown in the red bar, we found that speculation can effectively alleviate the penalty caused by ambiguity from the static binary analysis. However, if all memory accesses from the iteration of a loop are speculatively executed, the cost of memory redirection and speculative operations negate the benefit quickly. Therefore, the performance from speculation is highly sensitive to the transaction size. The larger transaction (big iteration size) from a loop, the higher slowdown it expects. The sensitivity is magnified by the mis-speculation rate, which is determined by the parallelism from the loop: the actual runtime data dependence existing from the undecided memory accesses. In conclusion, as the thread-level speculation is sensitive to its transaction size, in turn, it still has to rely on the accuracy of static binary analysis to prove and remove non-conflicting accesses from the transaction.

4.4 Related Work

We have seen huge efforts made in limit studies on *instruction-level parallelism (ILP)* [117, 118] hidden at binary level. Similarly, potentials for *thread-level parallelism* (TLP) have also been studied [119] extensively. However, TLP parallelism is much more complicated and problems vary at different granularities of the task. For the last three decades, a large stack of research has been studying these problems. Various granularities and task forms can be summarised as follows:

- Fixed-size block [120]
- Basic block [65] [1]
- Loop iterations or hot code[116, 121, 88, 122, 123]
- Procedure/function calls [121, 124, 125, 123]
- Pipeline stages [126]
- Generalised patterns [127]

From the above work, we observed that the most common task parallelism is *loop-level parallelism*, where task granularity is one or multiple loop-iterations, or more accurately, a dynamic sequence of instructions which are frequently executed. Each iteration of a loop can be executed by a speculative thread that runs in parallel with the other iterations of that loop. This dissertation studies the DOALL and DOACROSS loop level parallelism that can be easily transformed automatically without extensive binary analysis using synchronisation and TLS. Therefore only DOACROSS and TLS based limit studies are listed in this section.

Larus [114] studied the DOACROSS style limits on numeric and symbolic programs. His model analysed the execution traces and simulated the parallel execution patterns assuming unlimited parallel resources and zero threading cost. From his results, he concluded that massive speedup could be obtained from numeric programs but is not so good for symbolic programs.

Steffan and Mowry [116] investigated a preliminary limit study in their STAMPede Project. Firstly they manually selected hot loops from execution traces. Then they separated each loop iteration into forms of epochs, which are sequences of repeated instructions that were frequently executed. Data dependencies were obtained by analysing the traces. The speedup limit is calculated by assuming that all WAR and WAW dependencies could

[1]No limit studies

be resolved by renaming and each data forwarding RAW could be immediately consumed by the next epoch. They've found substantial parallelisation potential in their selected SPEC 92 and 95 benchmarks. However special hardware extension is needed to support the exploitation of the huge parallelisation potential.

Zhong et al [88] investigated the fraction of DOALL loops among SPEC benchmarks and compared results between statically-recognised DOALL and DOALL loops from runtime profiling. They showed that the fraction of actual DOALL at runtime is significantly larger than the recognised loops from static compilers. Even with sophisticated pointer and shape analysis, compilers failed to fully prove the actual data dependence pairs. Therefore they claimed there is much more parallelism obfuscated in the program which could only be exploited by thread-level speculation. Again, the simulation is based on the parameters from a hypothetical hardware extension to support speculation.

Von Koch et al [128] have the most similar limit study to the work in this chapter. They studied the upper bound of parallelism in the context of dynamic binary parallelisation and thread-level speculation. However, their results of limits do not reflect the true upper bound of parallelism in binaries. As there is a fraction of cross-iteration dependencies that can be removed by induction and reduction transformation, the mis-speculation rates in their evaluation could be reduced further and hence more parallelism could be retrieved. In contrast, BEEP relies on static binary analysis to identify opportunities that can remove data dependencies caused by clear induction and reduction operations. Moreover, BEEP is more flexible that combines more different parallel models with different assumptions using only instrumentation run.

4.5 Summary

This chapter presents a guided binary instrumentation framework called BEEP extended from the GBR platform. From the profiling results from BEEP, we investigate the limits of loop-level parallelism in SPEC2006 binaries with many parallel execution models under ideal and realistic assumptions.

In conclusion, this chapter answers three questions raised at the start of this chapter:

- How to retrieve sufficient information from binaries for parallelism limit studies: BEEP is designed to use a combination of static binary analysis and dynamic binary instrumentation. It achieves demand-driven binary instrumentation by using the GBR hint programs, where fine-grained instrumentation operations can be controlled from the static binary analysis. Moreover, high level global information of the program can be delivered through hint programs. Therefore BEEP is able to retrieve more high-level information and build more complicated parallel models to evaluate parallelism from complicated data structures.

- What is the theoretical upper bound of parallelism based on the dataflow and control flow nature from the machine code: three ideal parallel execution models are developed to investigate DOALL and DOACROSS loop parallelism only.

  - DOACROSS Dataflow Model: it is found most binaries do not exhibit enough parallelism if all data dependencies from the original binary are considered.
  - DOACROSS Induction/Reduction Model: a fraction of the loops see a substantial performance boost by relaxing the constraints of induction and reduction

variables. Many DOALL loops can be identified if induction/reduction dependencies are removed.

  - DOACROSS Code Motion Model: further performance improvement can be brought by rescheduling the instruction order of the loop iteration. This model suffers scalability issues in sorting and reordering of calculations.

- What is the estimated speedup given different realistic assumptions of parallelisation paradigms and real hardware under conservative static analysis: two realistic parallel execution models are developed based on conventional approaches for parallelisation.

  - Synchronisation Model: binary parallelisation using a pure synchronisation approach is insufficient to bring performance on conventional hardware. It is firstly limited by the extra synchronisation operations introduced by ambiguous static binary alias analysis. Secondly, the propagation cost is too high for current multicore systems to bring actual performance for DOACROSS-based parallelisation. It may only achieve a slight speedup when its L3 cache is guaranteed to hit during every data propagation.
  - Thread-level Speculation Model: speculation can effectively alleviate the penalty caused by ambiguity from the static binary analysis. However, the performance of the thread-level speculation is sensitive to its transaction size. To reduce the transaction size, it still has to rely on the accuracy of static binary analysis to prove and remove non-conflicting accesses from the transaction.

The next chapter builds from the above conclusions and discusses the implementation of automatic parallelisation in the GBR platform. The filtered loops from the profiling results are also selected for parallelisation on real systems.

Chapter 5

Automatic Binary Parallelisation Framework

With the binary recompilation infrastructure GBR discussed in chapter 3 and the ability to locate parallelism from binaries from chapter 4, now, we discuss the implementation of an automatic binary paralleliser in GBR. There are two different interpretations of "automatic":

1. The tool takes a legacy executable, performs heavy static binary analysis, generates a hint program and then directly performs parallelisation in the DBT. The whole flow of the analysis, hint program generation, and parallelisation is free of any manual assistance.

2. Only the dynamic transformation and parallelisation are automatic. It requires an existing correct hint program to be provided for the executable. This form of "automatic" does not require hints to be provided solely by our static binary analysis tool, but it can be generated by other static, profiling tools or manually hard-coded hints.

In this chapter, we aim for the first interpretation of "automatic", where the whole hint generation and parallelisation flow are free of manual intervention. Achieving full automatic hint generation may be challenging and requires significant engineering effort. If the static binary analysis is not able to provide automatic generation of hint programs for efficient parallelisation, we use profiling information to guide parallelisation, which falls to the second interpretation of automatic parallelisation.

We propose Guided Automatic Binary Parallelisation (GABP) by following the first interpretation, which is an automatic runtime system built on top of the binary recompilation engine GBR. GABP is implemented as an extension in GBR, along with other binary optimisation components such as automatic prefetching and vectorisation extensions. By sharing the same recompilation infrastructure, GABP relies on guidance from hint programs and performs binary recompilation at runtime. GABP assumes a fixed model for general loop structures during static analysis so that any transformation involving parallelisation of the loop can be decomposed into a set of fine-grained modification passes on each loop component. The combination of these modifications can provide complex functionality such as managing threads and enforcing correct data dependencies during parallelisation.

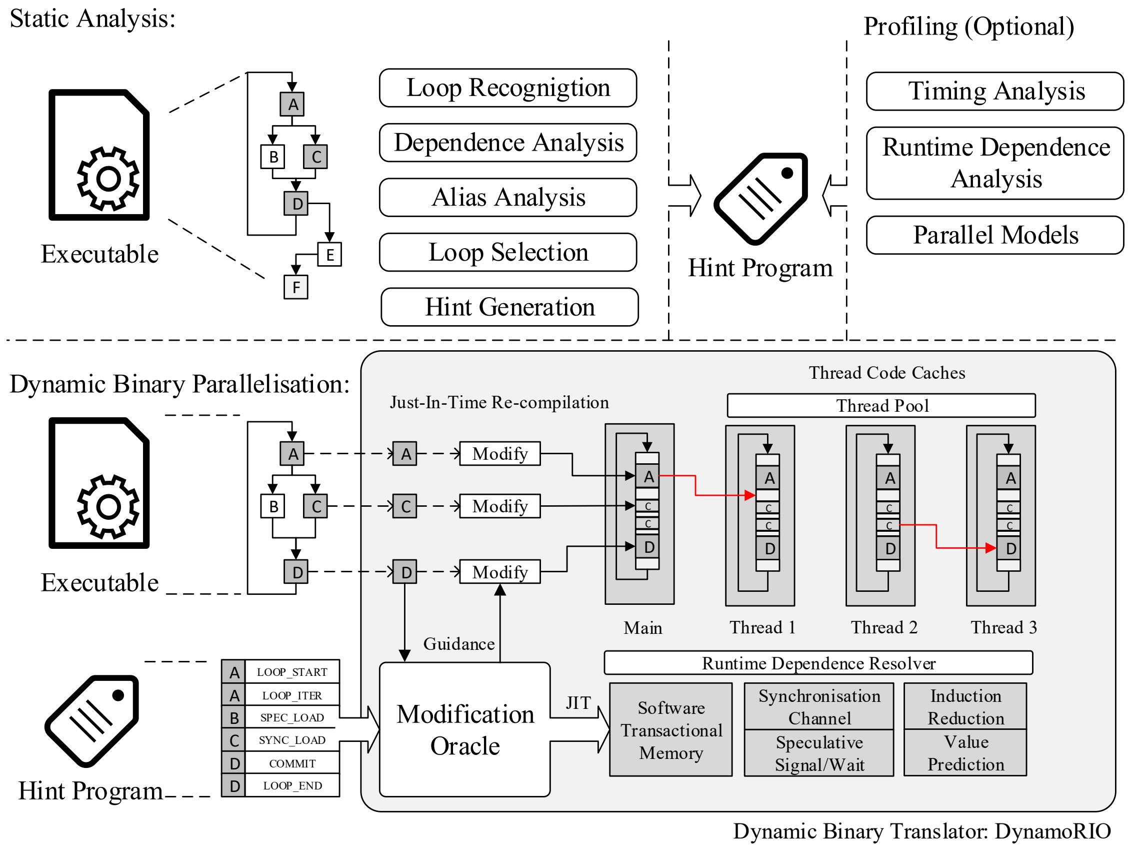


Figure 5.1: Overview of the Guided Automatic Binary Paralleliser (GABP)

The whole parallelisation problem is divided into two major factors: the *compilation* of a hint program from a given executable and the runtime *transformation* to enable parallel execution. In this chapter, I first describe how a hint program is compiled statically and then illustrate how the executable is recompiled under guidance from the hint program. We demonstrate that GABP is effective to achieve a geometric mean of 2.0x performance gain through parallelising SPEC2006 binaries on real hardware platforms.

5.1 System Overview

Figure 5.1 shows an overview of GABP. With the definition of hint program interfaces of GBR, the GABP system is divided into static and dynamic phases with three major components:

- `Static binary analyser`: loads the input executable, recognises, analyses, selects loops and generates hint programs.
- `Parallelism profiler` (BEEP): identifies the hot region (loop) of the executable and summarises frequent and rare cross-iteration dependencies in the loop.
- `Dynamic binary paralleliser`: interprets hint programs and parallelises while executing the application executable.

At the static phase, the static binary analyser takes a standard executable or a shared library binary as input. The input binary is disassembled, segmented and converted into an intermediate representation (IR) that contains all machine-level contexts. The IR is designed to be low-level while being slightly higher than the plain disassembly, which only abstracts register/stack and heap accesses into a universal variable representation while maintaining the original opcode for the instruction. From the disassembled IR, data and control graphs are constructed, and all the loops are identified. Among all recognised loops, only a fraction of the loops are selected for parallelisation according to a series of parallel cost models, which are discussed in chapter 4.

The static binary analysis is source-language agnostic, and does not require the availability of symbol tables or debugging information. Even with symbolic information from the source code, it is still difficult to reason about whether a loop is beneficial for parallelisation or even actually executes during a particular program run. We rely on profiling information from BEEP to help decision-making in static analysis for loop selection and handling potential runtime data dependencies.

Figure 5.2 illustrates the standard flow to enable automatic parallelisation for any given executable, which has three passes of static analysis and dynamic translation. Given an executable to be parallelised, the first step is to identify all loops with high coverage. To obtain the information, the static binary analyser generates coverage profiling hints in the hint program. The executable is then instrumented with training inputs, and a timer is dynamically generated at the beginning and finish of each recognised loop. After instrumentation, loops with low coverage are filtered out since they give low benefits regarding overall program speedup. The coverage information is sent back to the static binary analysis.

For the second pass, the static binary analyser further investigates the remaining loops with high coverage. For each loop, it generates BEEP instrumentation hints to evaluate parallelism. The instrumentation details are discussed in chapter 4. Instead of blindly storing massive traces or sampling all memory accesses from profiling, it only instruments the accesses which are not easily decided by static analysis. For example, Table 4.3 illustrates the profiling results for the SPEC2006 benchmarks and only those loops that are proved beneficial in realistic cost models are scheduled for parallelisation. BEEP generates a parallelism report for each loop that describes the statistics of runtime dependencies. For the final stage, the static analyser then re-evaluates the selected loops with the parallelism report and decides whether it is beneficial to parallelise the loop. Once the loop is selected for parallelisation, it is given a loop type for hint generation. Based on the loop type, a set of parallelisation hint instructions are generated and encapsulated in the hint program. The hint program can be kept throughout all subsequent parallelisation runs and the first two profiling passes can be avoided.

At the dynamic phase, the dynamic binary paralleliser is implemented through guided recompilation in GVM[1]. GVM interprets the parallel-related hint instructions from the input hint program and accordingly recompiles basic blocks from the selected loop into a parallel version. Moreover, GVM is independently invoked by the parallelising threads so that different versions of code are recompiled per thread from the same original code. The resulting recompiled thread-specific code is buffered in the respective thread-private code caches as shown in Figure 5.1. The thread-specific GVM implementation enables more opportunities for reducing the overhead for accessing thread-private variables instead

[1]Guided Virtual Machine, discussed in Section 3.3

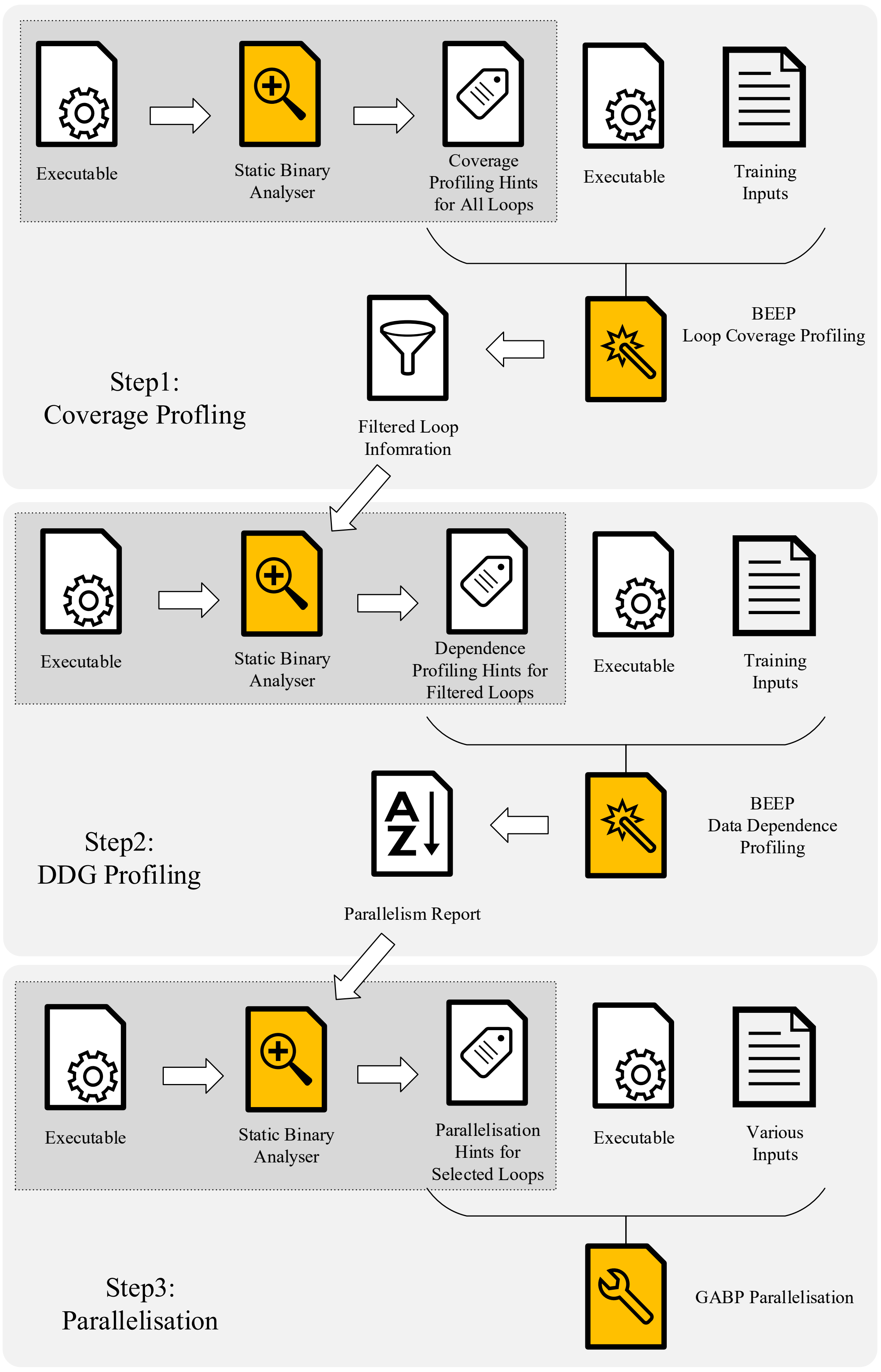


Figure 5.2: Standard flow to enable automatic parallelisation for any given executable

of frequently accessing thread ID extensively at runtime. In addition to recompilation support for threads, GABP also incorporates a runtime dependence resolver to address runtime data dependencies. The resolver consists of many code routines for privatisation, synchronisation, speculation and value prediction. Most of the routines are JIT compiled after the threading data structure is allocated and the number of threads is known. More details are discussed in Section 5.5.

5.2 Static Hint Generation

The main driving force for using hint instructions is to deliver static information to the program points that require modification when needed. Modification rules can be determined ahead of time and encoded as static hints. Hence they avoid the need to retrieve the information at runtime compared to other approaches such as runtime sampling and profiling. We divide hint instructions into two categories: event-based hints and information-based hints. Event-based hints are simply a specific set of annotated PC addresses in the binaries. When execution arrives at the annotated PC, GVM performs modifications based on the hint opcode. For example, event-based hints mark the exact addresses of loop starts, exits or the exact instruction that needs synchronisation between threads. Through event-based hints, the flow of parallelisation can be controlled.

Information-based hints provide information that GVM could not easily retrieve at runtime. For example, the information of current live registers, the size of stack frames, the iteration count for the current loop etc. The global static information can be encapsulated and delivered in information-based hint instructions. The information could not be easily obtained at runtime, because it has to be retrieved through a heavy-weight analysis on the global context of the program. Through information-based instructions, GVM is able to directly consume the fruits of global analysis of the program, which directly removes the need for runtime sampling to further retrieve information of interest.

The generation of both types of hint instruction must be safe and reliable, otherwise a tiny misguidance would result in incorrect recompilation of the original application. To maintain the consistency between the static and dynamic components, the static analysis must include an accurate model of GVM, with the same assumption on handling basic blocks, control flow and heuristics as the dynamic binary recompilation engine, so that the hint instructions can be correctly interpreted with correct runtime contexts.

5.2.1 Loop Recognition

After segmenting the input binary, the static binary analyser firstly identifies loops in the CFG by locating all the back edges through depth-first searches (DFS) in each procedure. The loop body can be identified by including all the nodes traversed throughout the cycle path in the CFG, or strongly connected components (SCC). A loop is called a *natural loop* [129] when there exists one and only one entry node `entry` that dominates the body nodes in the loop. Given a back edge $n \rightarrow e$, if node e dominates node n, it means the control flow must go through e in order to reach n.

With the definition of a natural loop with a single entry, it is possible to coalesce all loop body nodes into a single SCC. The parent CFG is simplified and the relations of loop nests can be constructed by analysing the positions of its entry nodes. The natural loop model is a strict loop model that guarantees the loop body exclusively belongs to the loop nests, which facilities exclusive transformation for the loop. If there exists more than one path that flow into the same cycle in the CFG, they are treated as unnatural loops and runtime checks must be generated to differentiate two different loops.

Once the entry block of a natural loop is identified, the sub-graph of the loop in the CFG fits into a loop model with the following constraints: a static natural loop L is a tuple of:

$$\begin{aligned} &\mathtt{L} \leftarrow (\mathtt{entry}, \mathtt{Init}, \mathtt{Body}, \mathtt{End}, \mathtt{Check}, \mathtt{Exit}) \\ \text{where: } &\forall \mathtt{n} \in \mathtt{End}, \mathtt{n} \in \mathtt{PRED(entry)} \wedge \mathtt{entry} \xrightarrow{\text{dom}} \mathtt{n} \wedge \mathtt{n} \xrightarrow{\text{backedge}} \mathtt{entry} \\ &\forall \mathtt{i} \in \mathtt{Init}, \mathtt{i} \in \mathtt{PRED(entry)} \wedge \mathtt{i} \notin \mathtt{End} \\ &\forall \mathtt{b} \in \mathtt{Body}, \mathtt{entry} \xrightarrow{\text{dom}} \mathtt{b} \wedge \exists \mathtt{n} \in \mathtt{End}, \mathtt{b} \in \mathtt{SCC(entry)} \\ &\forall \mathtt{x} \in \mathtt{Exit}, \exists p \in \mathtt{PRED(x)}, p \in \mathtt{Body} \\ &\forall \mathtt{c} \in \mathtt{Check}, c \in \mathtt{Body} \wedge \exists \mathtt{x} \in \mathtt{Exit}, \mathtt{c} \in \mathtt{PRED(x)} \end{aligned} \tag{5.1}$$

where the symbols refer to:

- $\mathtt{PRED(n)}$: the set of predecessors for node n in the CFG.
- $\mathtt{SCC(n)}$: the set of nodes that form a SCC that node n belongs to. The SCC is constructed according to Tarjan's algorithm [130].
- $\mathtt{m} \xrightarrow{\text{dom}} \mathtt{n}$: node n is dominated by node m.
- $\mathtt{m} \xrightarrow{\text{backedge}} \mathtt{n}$: a back edge originates from node m to node m.

The CFG nodes are pattern-matched and solved in the loop model. Figure 5.3 shows an example of the fraction of CFG matched according to the loop constraints. Once all constraints are met, the corresponding nodes are placed in different loop component buckets for further characterisation and analysis. If one of the constraints is not met, the loop is disabled for hint generation and thus not for further parallelisation. For all successfully recognised loops, a specific loop ID is assigned. The loop ID is used throughout all subsequent passes of static, dynamic and profiling analysis.

To maintain safety and correctness for further data flow analysis, all the instructions from the loop body are examined. Loops with undetermined indirect control flows, implicit changes of stack pointers, non-return subroutine/library/system calls, memory allocation, interrupts, exceptions and incompatible instructions or registers are rejected from further parallelisation. These rejected loops can be weakly accepted if they are verified as safe during profiling stages, such as verifying the shared library calls to be re-entrant and thread-safe.

5.2.2 Dependence and Alias Analysis

As all loops with undecided control flow are rejected, the remaining loops have fixed and deterministic control flow, making it easier to perform data flow analysis. From the disassembled instructions in the loop body, the disassembly is lifted into a custom intermediate representation. While all accesses to registers, stack elements and memory locations are abstracted to an intermediate structure called *loop variables*. Indirect memory accesses are also abstracted into wildcard loop variables. They are then analysed in the further

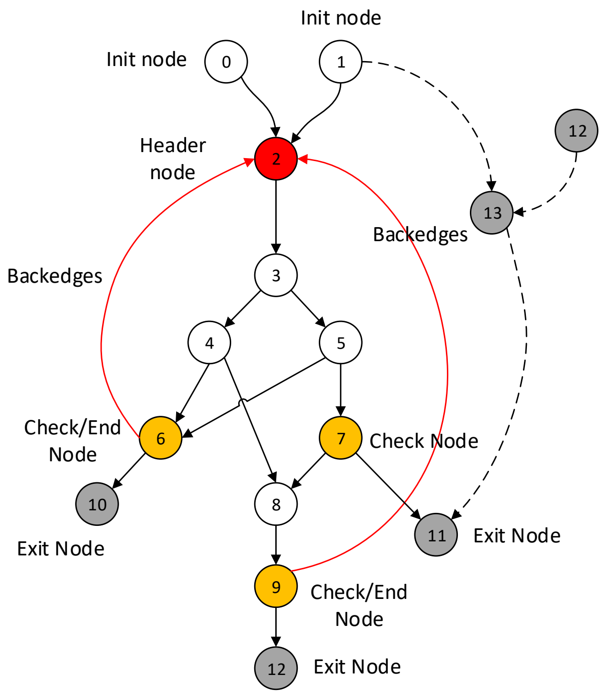


Figure 5.3: Tuple components for a typical natural loop.

alias analysis. Loop variables serve as the unit of storage for information propagated through the loop code. Once all the loop variables are abstracted, the following set of information is constructed:

- $\texttt{DEF}(\texttt{i})$: the set of loop variables modified by instruction `i`.
- $\texttt{USE}(\texttt{i})$: the set of loop variables read by instruction `i`.
- $\texttt{LiveIn}(\texttt{i})$: the set of loop variables that are live prior to instruction `i`.
- $\texttt{LiveOut}(\texttt{i})$: the set of loop variables that are live after instruction `i`.
- $\texttt{ReachBy}(\texttt{i}, \texttt{v})$: the set of instructions that generate the definition of loop variable `v` used by instruction `i`.
- $\texttt{ReachTo}(\texttt{i}, \texttt{v})$: the set of instructions that use the definition of loop variable `v` generated by instruction `i`.

Compared to conventional compiler-based analysis, a few corner cases on subroutine calls need to be considered. For a given `call` instruction, the static tool further analyses the subroutine if the subroutine is a leaf function. The `DEF` and `USE` set of the function coalesces into the parent `call` instruction. For other complex and nested subroutine calls or shared library calls, we assume a standard calling convention on function calls. The external function is assumed to use and define all the argument and return registers. Once the liveness and reaching information have been constructed, all loop variables are divided into three categories with the following conditions:

- `ReadOnly`: $\texttt{v} \in \bigcup_i^L \texttt{USE}(\texttt{i}) \wedge \texttt{v} \notin \bigcup_i^L \texttt{DEF}(\texttt{i})$

- Depending: $\mathtt{v} \in \bigcup_i^L \mathtt{DEF(i)} \land \mathtt{v} \in \mathtt{LiveIn(entry_L)}$
- Private: $\mathtt{v} \in \bigcup_i^L \mathtt{DEF(i)} \land \mathtt{v} \notin \mathtt{LiveIn(entry_L)}$

ReadOnly variables are not modified across the whole loop. Therefore they are not specifically handled during parallelisation. Depending variables are actually cross-iteration dependencies, since they are live across the entry of the loop. For each iteration, the access pattern for Depending variables is always a first read in at least one path in the loop body followed by at least one write later. There may exist more reads or writes between the first read and last write. Depending variables must be removed or resolved between threads by using runtime techniques including value prediction, synchronisation or speculation. If a variable is defined in the loop but not live across the entry of the loop, it is a Private variable which can be privatised to local write buffers during parallelisation.

A Depending variable can be further categorised based on its modification pattern in the loop.

- Induction: v is updated once along all paths throughout the loop and it is added or subtracted with the same constant offset for each path in the loop.
- Reduction: $\mathtt{v} \in \bigcup_i^L \mathtt{DEF(i)} \land \mathtt{v} \in \mathtt{LiveIn(exit)}$ and v is updated using closed form of accumulation such as addition or subtraction.

In practice, there exist a large fraction of memory operands which cannot be easily reasoned whether they are alias on a single loop variable. We perform a simple alias analysis on these memory operands. For example, a typical x86 operand is in the form of [base + offset *scale + disp], where base and offset are general purpose registers and they are abstracted into loop variables.

- If base is a ReadOnly variable and there is no offset register, the memory operand can be lifted to a normal loop variable and scheduled to normal liveness and reaching analysis across the loop.
- If base is a ReadOnly variable and the offset register is the original or a direct descendant of an Induction variable. It is a Private variable whose value is related to the loop iteration.
- If base is reached by a LEA[2] instruction in the loop, and the source of the LEA instruction is a ReadOnly variable. It is similar to the case when base is a ReadOnly variable.
- If base is the register RIP, it is a PC relative operand that represents an absolute memory location. The operand can be lifted to a normal loop variable and scheduled to normal liveness and reaching analysis across the loop.

All the other complicated combinations and calculations of memory addressing modes are labelled as "undecided" memory accesses. Figure 5.4 shows the degree of memory ambiguities found in the discussed implementation of static binary analysis for the SPEC2006 benchmark. It is found that around 22% of total loop instructions are memory accesses that can't be easily decided for their *alias* properties. There is existing work [40, 41] that

[2]Load effective address in x86, typically used in pointer arithmetic calculation.

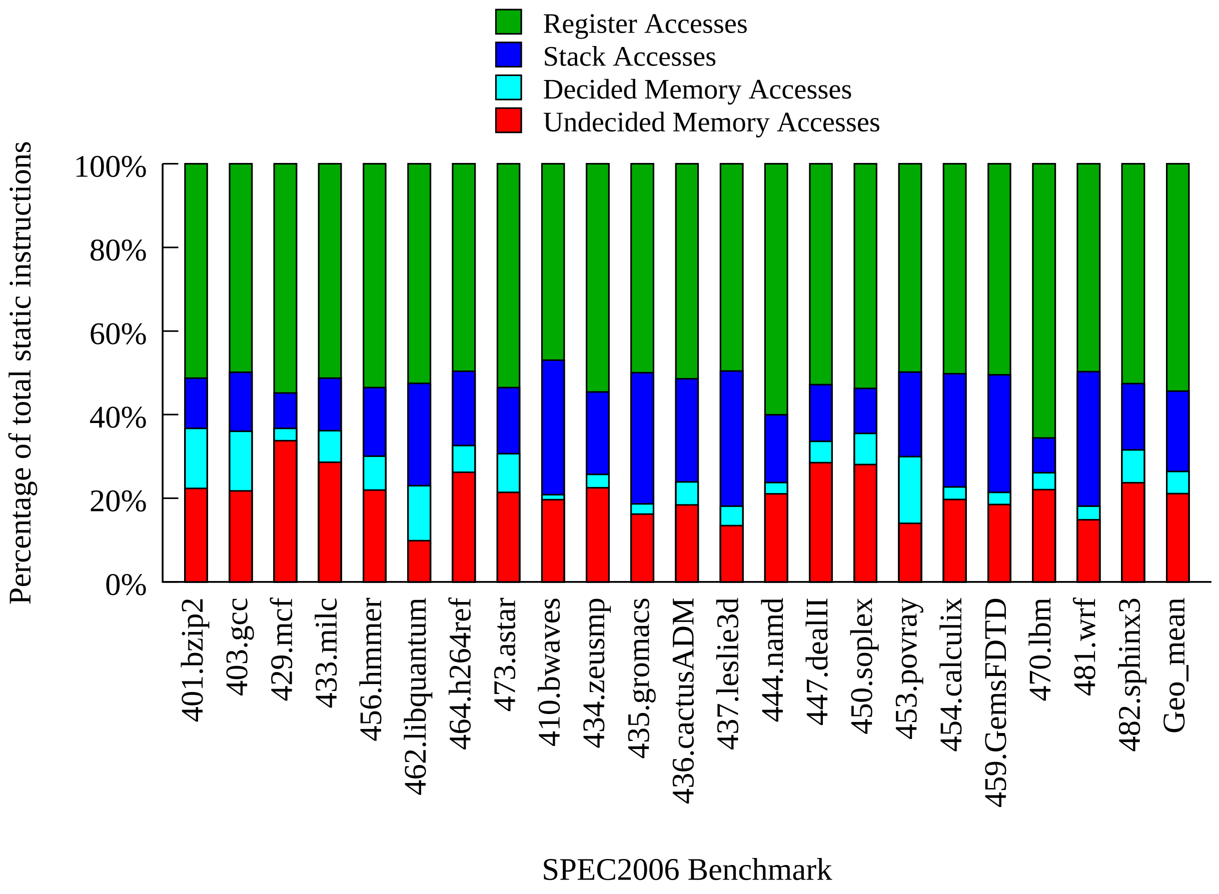


Figure 5.4: Limitation of static analysis: the proportion of undecided memory accesses for cross-iteration dependencies among all loop instructions. The static binary analysis implementation is based on the description in Section 5.2.2.

improves the accuracy of static alias analysis through heavyweight data-flow analysis. However, the results are not adequate to address all alias relations. Instead of pushing the accuracy of static binary analysis even higher, GABP relies on profiling of these undecided accesses during the profiling stage. By inserting hint instructions at these locations, a guided binary instrumentation is performed specifically on these accesses. The results of the profiling help the static analysis to relabel these wildcard variables as either ReadOnly, Depending or Private variables.

The absence of cross-iteration dependencies during profiling is not a guarantee that a loop is safe to parallelise for all inputs. Therefore GABP relies on additional thread-level speculation techniques to detect the cases when the data violations do occur. All the statically-undecided memory accesses in a loop are surrounded in a software maintained transaction. Their read accesses are checked before the transaction is allowed to commit. Therefore, a correct parallel execution can be guaranteed using runtime speculation regardless of the inputs, while it may not be optimal in terms of performance due to potential rollback costs.

Using profiling information to assist static analysis is still an intermediate stage before a full comprehensive static binary analysis is introduced. Due to timing and cost constraints, the implementation of a much more complex static alias analysis for binaries is treated as another research topic in the future.

5.2.3 Loop Characterisation and Selection

The final step for hint generation is to select the most beneficial loops for parallelisation. As discussed in chapter 4, it is infeasible to select beneficial loops purely though static binary analysis, as the hot region of the executable could only be obtained through profiling. Therefore two profiling passes are performed before actual loop selection for parallelisation, as shown in Figure 5.2. The timing coverage, the estimation of average iteration count, total invocation count, graphs of cross-iteration dependencies and estimation of the speedup is retrieved from the parallelism report derived from BEEP profiling.

To automatically select the most beneficial loops from binaries, firstly the loops with less than 1% of coverage or with less than 1.1x ideal speedup estimation are filtered out of parallelisation, since they never deliver effective overall program performance improvement alone. Secondly, for the remaining beneficial loops, intra-loop nests are built based on the call graph and control flow relations. As it is not able to parallelise both outer and inner loops at the same time, only one loop from a loop nest is selected for parallelisation. The loop with highest effective overall program speedup is selected for parallelisation.

Lastly, it performs final checks on the prospective loop candidates by verifying its cross-iteration dependencies statically. If the profiling dependence information is available, the static analysis also verifies its analysis result with the profiled data dependencies. The profiled dependence graph should be a subset of the static dependence graph which is conservative to include all undecided patterns. If there exists a profiled dependence pair that disagrees with decided static dependencies, the loop would be subject to manual investigation. For each selected loop, the static tool determines the type of loop and its corresponding parallelisation policy.

Currently, GABP supports four different types of parallelisation.

- `DOALL_Block` loop: there are no other cross-iteration dependencies except `Induction` variables and `Reduction` variables with `add` and `sub` reduction operation. The total iteration count can be determined at the entry of the loop. There is no `break` statement in the loop, where all loop terminations only occur based on the check of induction variable against the constant iteration count.

- `DOALL_Cyclic` loop: there are no other cross-iteration dependencies except `Induction` variables and `Reduction` variables with `add` and `sub` reduction operation. The total iteration count cannot be determined at the entry of the loop or there is `break` statement in the loop, where there exist loop exits that depend on checks on the conditions that are irrelevant of iteration count.

- `DOACROSS_Synchronisation` there are clear and few cross-iteration dependencies besides `Induction` variables and `Reduction` variables. The occurrence of all cross-iteration dependencies are frequent.

- `DOACROSS_Speculation` there are cross-iteration dependencies besides `Induction` variables and `Reduction` variables. The occurrence of most cross-iteration dependencies are rare per iteration.

Currently GABP does not support other types of parallelism such as `Pipelined` parallelism, due to its significant scale of code modification on control flows and its complexity in recompiling binaries.

Hint Generation Location	Loop Type For Parallelisation			
	DOALL_BLOCK	DOALL_CYCLIC	DOACROSS_SYNC	DOACROSS_SPEC
	Event-based hint instructions			
∀b ∈ Init, Tail(b)	LOOP_START_DOALL	LOOP_START_DOALL	LOOP_START	LOOP_START
∀b ∈ Init, Tail(b)	SCHED_THREAD	SCHED_THREAD	SCHED_THREAD	SCHED_THREAD
Head(entry)		UPDATE_INDUCTVAR	EXECUTE_SEQSEG	TX_START
∀b ∈ End, Tail(b)				TX_COMMIT
∀b ∈ Check, cmp(b)	UPDATE_CHECK			
∀b ∈ Body, dep_read(b)			SYNC_WAIT	SPEC_READ
∀b ∈ Body, dep_write(b)			SYNC_SIGNAL	SPEC_WRITE
∀b ∈ Body, private(b)	PRIVATISE_VAR	PRIVATISE_VAR	PRIVATISE_VAR	PRIVATISE_VAR
∀b ∈ Exit, Head(b)	LOOP_FINISH_DOALL	LOOP_FINISH_DOALL	LOOP_FINISH	LOOP_FINISH
∀b ∈ Exit, Head(b)	YIELD_THREAD	YIELD_THREAD	YIELD_THREAD	YIELD_THREAD

Table 5.1: Locations and major event-based hint instruction generation for parallelising four types of loops, where Head(b), Tail(b) refer to the location before the head and after the tail instruction of the basic block b respectively. The cmp(b), dep_read(b), dep_write(b), private(b) are the set of instructions in basic block b that contain compare, depending and private variables respectively.

5.2.4 Hint Generation

Based on the loop type, different sets of hint instructions are generated at the corresponding loop tuple component (entry, Init, Body, End, Check, Exit). Table 5.1 shows the list of typical event-based hint generation model for the four parallel loop types. The static hint generation must conservatively cover all possible combinations of runtime paths. Each single Init and Exit node must be annotated by the corresponding hint instruction, so that all loop entries and exits are fully under control by GABP. Moreover, each hint instruction is assigned with a priority. If two hints are annotated on the same location, the hint instruction of higher priority is firstly interpreted by GVM.

Each hint is also accompanied by a runtime guard, so that the hint instruction can generate runtime checks to protect its modified code. For some modifications, it is effective only when a certain runtime condition is satisfied. For example, if there were a path between two exit nodes of the loop, only the first exit encountered at runtime should finish the loop and yield the thread. The second exit node should be treated as void.

The groups of hint instructions are stored in a structured hint program. Besides hint instructions, a hint program header is included to encapsulate related meta information for just-in-time compilation and loop parallelisation. The detailed specification of the hint program header and hint instruction ISA is listed in Appendix B.

5.3 Thread Management

This section discusses the dynamic phase of GABP that realises binary parallelisation. When a hint program is successfully generated by the static binary analyser, it can be directly loaded into GVM for recompilation at runtime. The hint program can be reused indefinitely as long as the underlying binary does not change. The foundation of the dynamic paralleliser in GABP is to interpret hints in a deterministic parallel system and generate the same and correct output as the sequential execution.

Under the hood, threads are seamlessly created and deleted as the sequential binary is being recompiled under the GBR framework. In order to reduce the cost of thread spawning and increase the responsiveness for scheduling, threads are created whenever

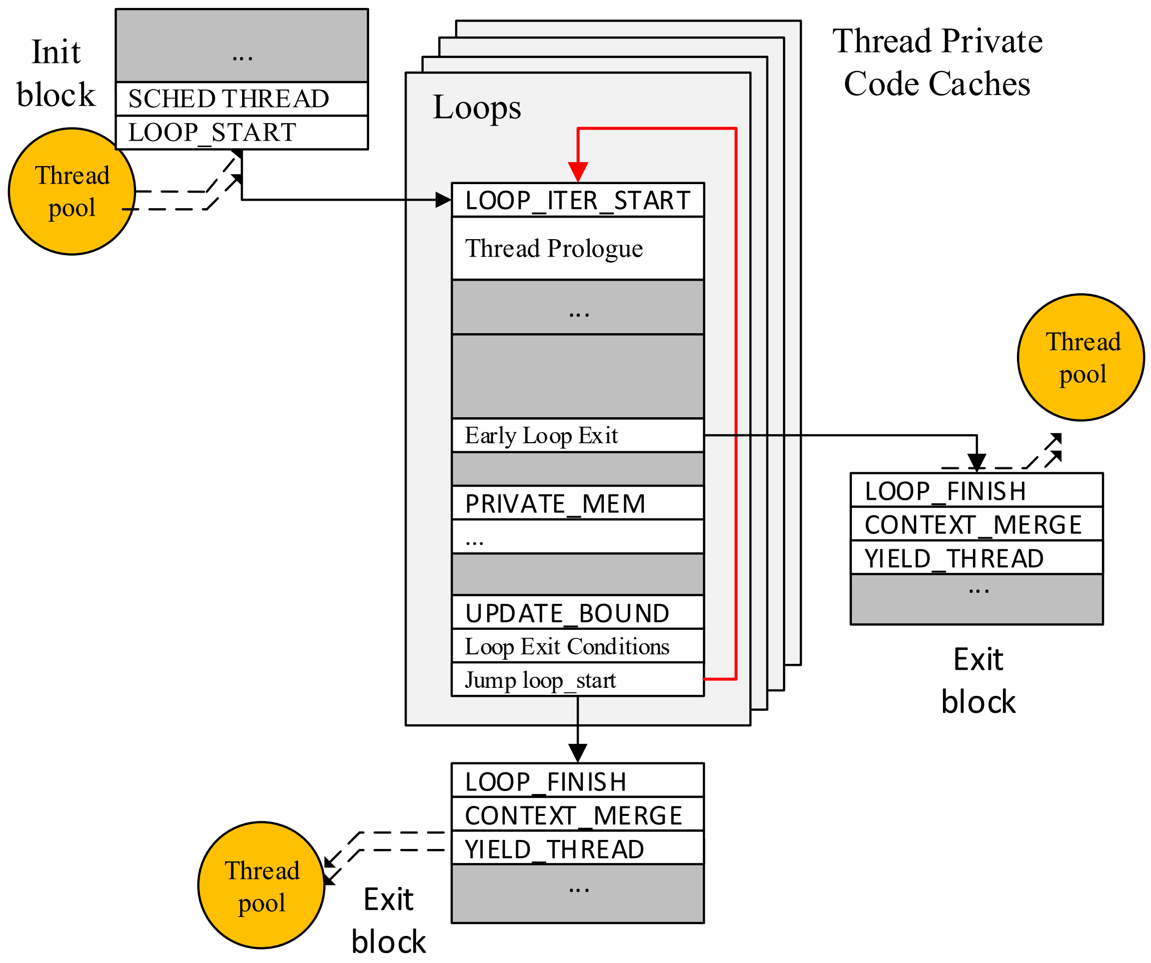


Figure 5.5: Example: Hint instructions are annotated on the corresponding location of the DOACROSS_SPEC loop to enable thread-level speculation.

the application starts running and placed in a thread pool waiting for commands. Once there is a need for parallelisation, they are scheduled to execute the designated code. After parallelising tasks are completed, threads jump back to the thread pool and wait to be re-scheduled. However, in practice, the whole control process is not an easy task in the environment of dynamic binary translation on real hardware and operating systems.

5.3.1 Threading States

GABP addresses the thread management problem by dividing all allowed thread environments into a finite set of states that form a finite state machine (FSM). A set of locks is inserted to prevent potential race conditions between thread operations. These locks are inlined dynamically at the locations specified by the static binary analysis through hint instructions.

As discussed in Section 2.1.1, based on the nature of dynamic binary translation, the thread states can be divided into two phases: `Transformation` and `Execution`.

- `Transformation` phase: When threads are in a `Transformation` phase, they are fully isolated from the original application context. The `Transformation` phase includes GVM routines for discovering code, interpreting hint instructions, transforming, compiling and buffering the modified code in the code cache. Execution of the application code should not be allowed in the `Transformation` phase.

- `Execution` phase: when threads execute instructions natively from the respective code cache or dedicated pre-compiled code snippets within the machine context for the application, they are in the `Execution` phase.

Migration between the `Transformation` phase and `Execution` phase must be performed through a full context switch. When threads are in `Execution` phase, they must not perform direct jumps to function routines that belong to `Transformation` phase and vice versa. The transformation phase is only active when a new piece of application code is discovered, while most of time is spent in the execution phase. Therefore thread states in the transformation phase can tolerate overheads from highly complex recompilation code.

In GABP, parallel threads are controlled in the FSM by five major states. The state transition diagram is shown on the right of the Figure 5.6. Note that most of the thread states are put in the `Transformation` phase to avoid duplicated translation of thread control code to its code caches.

- `Init` state: after threads are created, they are in the `Init` state. Threads perform initialisation on their respective thread local storages (TLS), including privatisation buffers, software transactional routines, synchronisation channels and linking the TLS of neighbouring threads. The majority of `Init` tasks can be performed in the `Transformation` phase.

- `Pool` state: when the main thread has not yet reached the loop code, other parallel threads are idle and spinning in an infinite loop, representing the `Pool` state. When threads are in the `Pool` state, they should be in the `Transformation` phase since there is no need to waste resources on dynamic translating the pool state code into code caches. A thread is nominated as a *warden thread*. The warden thread can JIT generate utility routines that are going to be used by other threads.

- `Start` state: it refers to the state when the main thread reaches the loop `Init` block annotated by the `LOOP_INIT` hint instruction. The main thread saves the current register state to a shared context structure. Each thread leaves the thread pool, replicates the main thread's context by performing a context switch from the saved shared context. It then executes JIT generated code to predict initial values for each `Depending` variable including induction and reduction variables.

- `Work` state(s): parallel threads execute their respective iterations from the loop. Each thread interprets hint instructions, translates the application independently by modifying the same original code to its own private version and buffers the code in its corresponding code cache. Depending on the selected parallelisation scheme, there are one or more sub-states: synchronisation states such as sequential segments; thread-level speculation states such as transaction start, validation and commit.

- `Finish` state: it refers to the state when any of the parallel threads reaches the `Exit` block of a loop annotated by the `LOOP_FINISH` hint instruction. Each thread commits its register and stack states to a thread private buffer and waits for the main thread to collect and merge them into a correct final application context. The

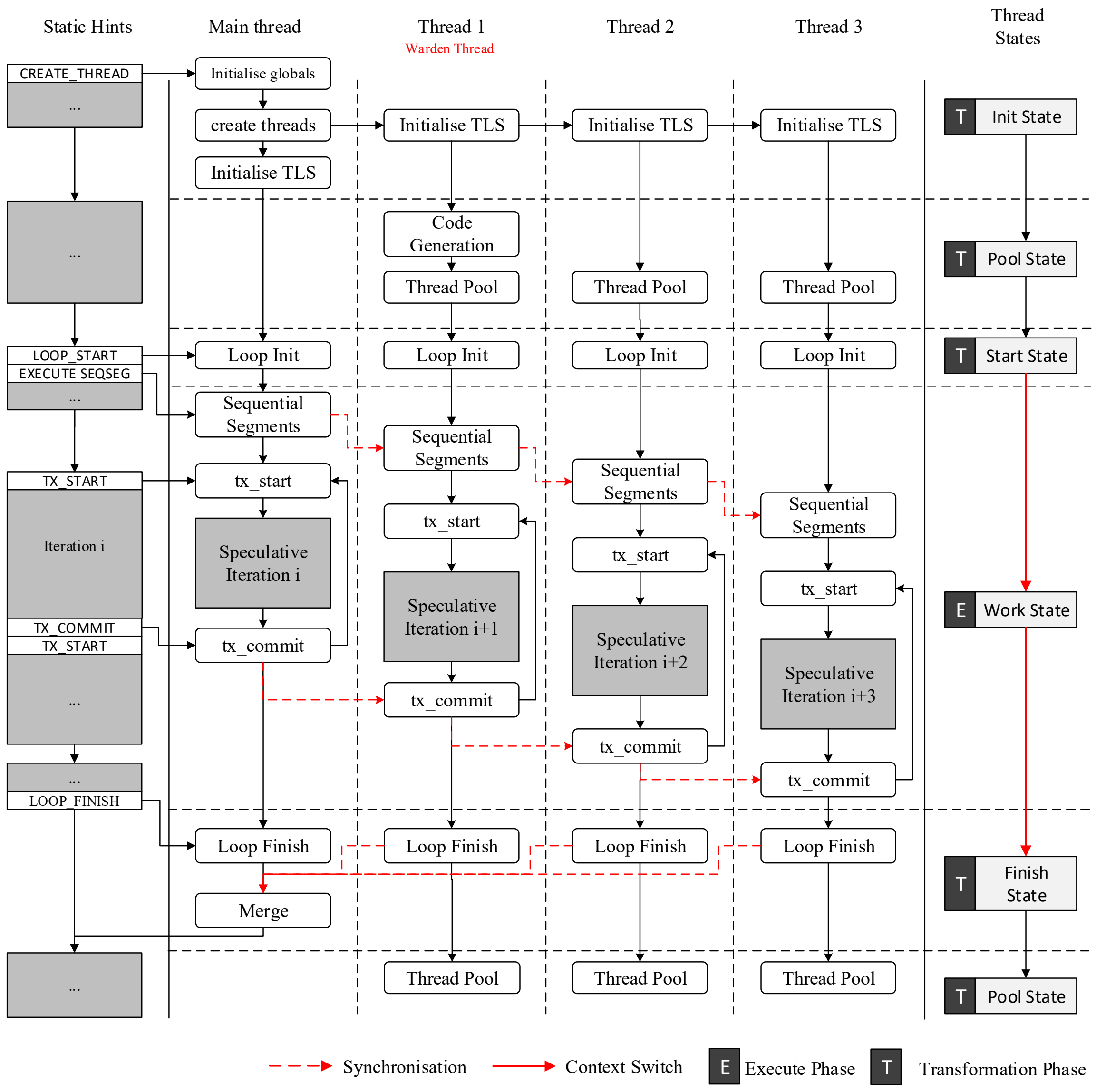


Figure 5.6: Thread states generation is guided by the annotated hint instructions.

merge process includes the calculation of `Reduction` variables from thread private copies. The final version of privatised variables from threads are also selected for thread merging. Afterwards, the main thread performs a context switch to the loop `Exit` block and resumes normal sequential execution.

To prevent potential race conditions during parallelisation, all parallel threads must be in the same state and meet the same flag conditions in order to propagate to the next state. Although it introduces more overhead in thread management, it provides deterministic outputs on a real system with weak sequential consistency memory models. In practice, threads might be de-scheduled or moved to other cores by the OS during any state of the FSM. It is essential to perform checks on all five stage transitions to prevent deadlocks and race conditions. For example, if there were no checks from the `Finish` state to the `Pool` state, it is likely that threads may work on different invocations of the loop at same time. Here we list a few synchronisation points and flags during the state

Pseudo Lock Code

```
//main thread waits for
//all other parallel threads
for (i=i; i<num_thread; i++) {
  while (tls[i].finish != 1);
}
```

JIT generated code

```
a5e2c001: cmpl $0x1,-0x5581fa83(%rip)
a5e2c00b: jne  0x55e2c001
a5e2c011: cmpl $0x1,-0x5581f543(%rip)
a5e2c01b: jne  0x55e2c011
a5e2c021: cmpl $0x1,-0x5581ece3(%rip)
a5e2c02b: jne  0x55e2c021
```

Figure 5.7: The main thread checks each thread's private locks. The checking code for the main thread is JIT generated after the input is determined `num_thread` = 3 and all `tls`[`i`].`finish` are initialised.

transitions:

- `Pool` → `Start` (`run` flag): the main thread only sets the `run` flag when all parallel threads are in the thread pool.
- `Start` → `Work` (`valid` flag): parallel threads are only allowed to perform initialisation after the main thread replicates its context to a shared context buffer.
- `Work` → `Work` (`canCommit` flag): parallel threads are only allowed to commit and start work on an another iteration when the previous threads are finished[3].
- `Work` → `Finish` (`finish` flag): the main thread only performs merges after all parallel threads commit their contexts to private buffers.
- `Finish` → `Pool` (`inPool` flag): parallel threads are only permitted to enter the thread pool when the main thread finishes merging.

Based on the many patterns of synchronisation, three types of locks are used.

- `Shared` lock: updated by the main thread, checked by parallel threads.
- `Peer` lock: updated by the previous thread, checked by the current thread.
- `Private` lock: one per parallel thread, updated by parallel threads respectively, all checked by the main thread.

All three types of lock are assigned with their own hint instruction. They are inserted and inlined at specific binary locations according to the specification of hint instructions. The lock code is JIT generated as simple absolute accesses since their dynamic mutex addresses are determined during generation of the lock code. The code is inlined into the original instruction stream with minimum register pressure. Figure 5.7 shows an example of the JIT generated lock.

[3]Serial commit constraints can be relaxed for DOALL loops

5.3.2 Thread Privatisation

Implementing the threading states ensures the correct and deterministic control during parallelisation. The second thread management problem is to maintain correct data isolation and privatisation for threads though the thread local storage (TLS). Privatisation is heavily used throughout the parallelisation. It is also the foundation to realise further runtime dependence handling techniques such as synchronisation and thread-level speculation.

5.3.2.1 Thread Local Storage

Implementing the thread local storage for parallelisation under the environment of dynamic binary translation is a challenging task. There are two major challenges faced to support TLS under DBT.

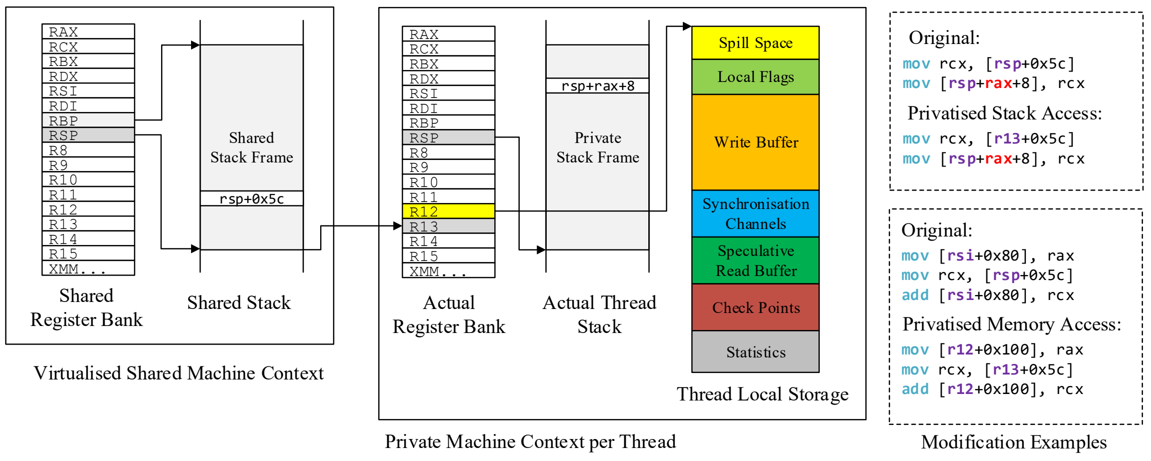
The first challenge is the conflict uses[4] of the same hardware threading location between existing threading libraries such as the `pthread`[3] and the transparent DBT such as DynamoRIO libraries. Therefore, I implemented a custom threading library instead of using the `pthread` library. The custom threading library directly calls the `clone` system call in Linux and avoids conflicts in TLS allocation and stack management from DynamoRIO.

The second challenge is the vast performance penalty on frequently accessing conventional TLS. Usually the cost of using thread-local variables can be ignored outside of a loop. However, if the thread-local variable is accessed very frequently in a hot loop, the cost may become an issue. Most compiler implementations support thread local variables declared with the `__thread` attribute. The implementation of the `__thread` variables is typically through a TLS lookup table for each thread. Accessing each `__thread` variable converts to a dynamic hash table lookup by calling a shared library call `__tls_get_addr`, which normally takes hundreds of cycles.

In GABP, the performance of accessing TLS is significantly improved by two approaches: one is to encode direct addresses of thread local variables in the respective thread-private code caches. The addresses of the TLS variables are obtained during the `Transformation` phase through normal hash table lookup for the thread variables. The returned addresses are then translated as immediate accesses such as PC-relative accesses to be executed in the `Execution` phase. Subsequent execution in the code cache can completely avoid the hash table lookups. All access to TLS variables becomes normal memory accesses.

However, encoding PC-relative instructions typically results in a large translated block size. Its referencing offset is also limited. The alternative approach, is to steal a general purpose register from one of the least-used registers in the loop and permanently buffer the TLS header during the execution of the loop. The least-used register information is provided by hint instructions, where the information is obtained from the global liveness analysis during static binary analysis on the loop. Both approaches reduce the overhead of TLS accesses to normal memory accesses during execution of hot loops. As encoding PC-relative accesses in the first approach may result in larger size in the final compiled

[4]Both `pthread` and `DynamoRIO` may use the same segment register `fs` (x86-64) to store the TLS and DynamoRIO context respectively. Therefore `pthread` is not well supported by DynamoRIO for the Linux OS.



Original:

```
mov rcx, [rsp+0x5c]
mov [rsp+rax+8], rcx
```

Privatised Stack Access:

```
mov rcx, [r13+0x5c]
mov [rsp+rax+8], rcx
```

Original:

```
mov [rsi+0x80], rax
mov rcx, [rsp+0x5c]
add [rsi+0x80], rcx
```

Privatised Memory Access:

```
mov [r12+0x100], rax
mov rcx, [r13+0x5c]
add [r12+0x100], rcx
```

Figure 5.8: The original machine context is virtualised in memory.

code, the second approach is preferred if there are frequent accesses to the TLS. However, if there is no free register to steal, the first approach can be used.

5.3.2.2 Register and Stack Privatisation

The hardware registers and stack space for each core is the most suitable and fastest space to store privatised variables. By doing this, the original application machine context has to be saved in an another place. To maintain data consistency during thread privatisation, a shared machine context structure is virtualised in memory. The shared machine context structure should always represent the shared, committed and correct states for the original sequential execution. Any writes to the shared machine context must be in a critical section surrounded with locks. When a thread finishes its task, it commits the changes from its private registers and stack locations to this shared structure. It is also the space for threads to communicate and enforce cross-iteration data dependencies from registers and stack elements. Since it always represents the correct value of the execution, it is also used by the speculation read validation if the thread-level speculation is enabled. More details are discussed in Section 5.5.1.

When the execution reaches the tail of the `Init` block of a loop, the main thread spills all the contents of its general purpose registers to the shared register bank. If the vector or floating point registers are also used in the loop, they are also spilled to the bank. Once the registers are ready in the shared machine context, each thread is able to only copy a selection of registers to its own private registers during loop initialisation. Each thread only copies the registers that are read during the whole execution of the loop. And the code for selective register copying is JIT-compiled by the warden thread.

While for stack elements, it might be expensive to copy all the stack elements to each thread's private stack, especially for applications with large stack frames or stack allocation that does not follow standard calling conventions. Therefore, the original stack used by the main thread is protected and shared across by all the parallel threads. And each thread allocates its own private stacks. To access the shared stack, another least-used register is picked by the static analysis. The selected register is used as the "shared stack pointer" that points to the protected stack frame throughout the loop execution. Each

```
Original                  Undecided Access          Decided Absolute

add [rsi+0x80], rax       lea r8, [rsi+0x80]        add [tls+0x100], rax
                          mov r9, r8
                          and r9, 0xffff
                          cmp [wbuf, r9], r8
                          jne probe_more
                          mov r10, [wbuf,r9,8]
                          add [r10], rax
```

Figure 5.9: Inlined heap privatisation. If the memory access is undecided, a hash function is used. All free registers are determined statically and the inline code is generated dynamically. If the memory access is decided and identified as private, the hash table lookup can be avoided. Additional initialisation is required to copy to each thread.

thread can access the shared stack for read-only stack elements and use private stack for temporary variables.

For example, Figure 5.8 illustrates the structure of the shared machine context and thread-private machine context. The register `R13` is picked to store the original stack pointer. A `ReadOnly` stack element such as `[rsp+0x5c]` is modified into `[r13+0x5c]` so that it reads from the main stack instead of the thread's stack. A `Private` stack element such as `[rsp+rax+8]` remains unchanged, which results in a direct privatisation for the write of the stack element. Different privatisation modification is generated according to the type of the stack element whether it is `ReadOnly` or `Private`. The type of each stack element is provided by a specific hint instruction `PRIVATISE_STACK`, which is generated based on the dependence analysis algorithm discussed in Section 5.2.2.

In GABP, it requires the main thread to perform equal tasks as other parallel threads. To prevent the main thread from polluting the temporary variables stored in the original stack, the main thread also need to switch to a new private stack, while leaving the original main stack protected and shared across all parallel threads. It switches back to the original stack after the loop finishes after the merge operation in the finish state.

5.3.2.3 Heap Privatisation

As heap accesses are unbounded and scattered in the whole memory address space, it is expensive to duplicate and privatise the whole heap address space for each thread. As the result, all heap accesses must be hashed into bounded thread local write buffers. Figure 5.9 illustrates the process of hashing for undecided memory accesses. The dynamic address of the memory accesses `r8` is calculated and the last few bits are retrieved with a hash mask by the instruction `and r9, 0xffff`. The resulting `r9` is treated as the key for the hash table.

The hash table is implemented as a linear-probe mapping between the original memory address and the redirected memory location in the write buffer. If the query address gets hit in the table, the redirected address is directly loaded and used for direct accessing. For example in Figure 5.9 (middle), the redirected address is loaded into `r10` from the hit table entry `[wbuf,r9,8]`. The register is directly replaced with the original memory operand `[rsi+0x80]`. If the query address gets a miss, it calls `probe_more` that iteratively checks the stored memory address in the next entry. If an empty entry is found, a new entry is created by loading the input address and data in the entry. The allocated entry

Example loop source code:

```
//loop 16 in quantum_toffoli in 462.libquantum
for(i=0; i<reg->size; i++) {
  if(reg->node[i].state & ((MAX_UNSIGNED) 1 << control1))
    if(reg->node[i].state & ((MAX_UNSIGNED) 1 << control2))
      reg->node[i].state ^= ((MAX_UNSIGNED) 1 << target);
}
quantum_decohere(reg);
```

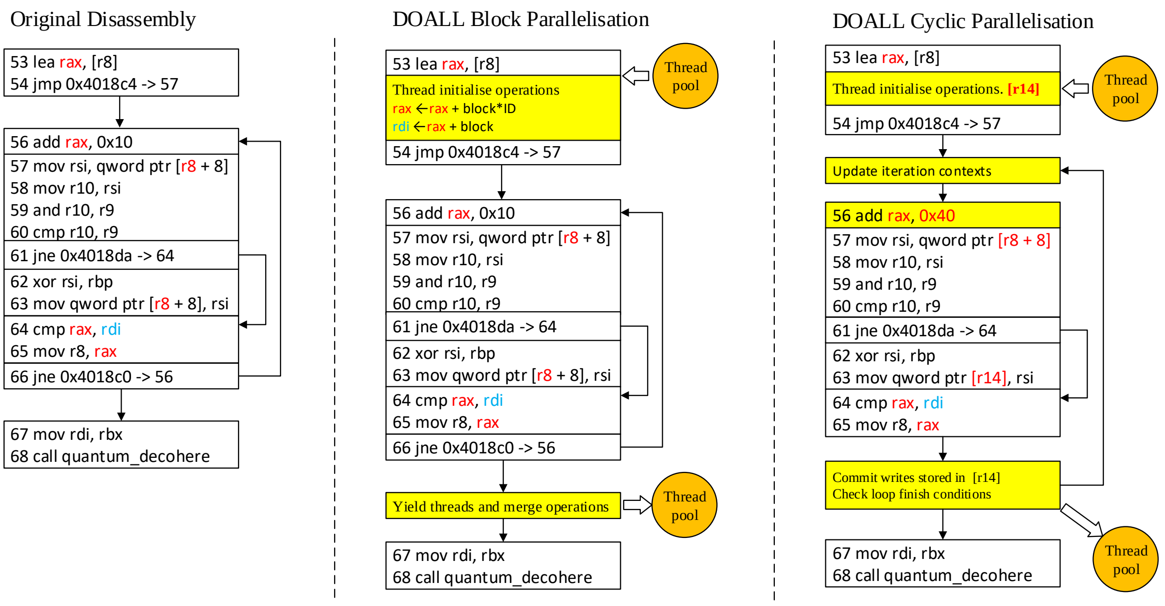


Figure 5.10: Block and cyclic DOALL parallelisation.

is then returned in register `r10`.

Privatising a large number of undecided heap accesses would increase the chance of collisions for the linear-probed hash table, which results in a huge performance penalty. One effective approach to reduce the hash table lookup is to directly encode the memory operand to the thread local storage. If it statically proves that the instruction is only accessing one dedicated memory address and there are no other pointers that refer to the address, a dedicated thread local storage can be used to redirect the fixed address. Hash table lookup can be replaced with a single TLS memory access, which is shown in the second example in Table 5.9. This benefit is only brought by the combination of static analysis and JIT compilation.

5.4 DOALL Loop Parallelisation

With the ability to deterministically control the thread states and isolating thread contexts using guided privatisation, it is then feasible to realise the automatic parallelisation of DOALL loops where there are no cross-iteration dependencies. From the discussion in Section 5.2.3, the static tool generates two sets of hint instructions for the `DOALL_Block` loop and `DOALL_Cyclic` loop respectively.

5.4.1 Block Parallelisation

A loop is labelled as `DOALL_Block` when there are no other cross-iteration dependencies except `Induction` variables and `Reduction` variables with a simple reduction operation. Parallelism is achieved by scheduling threads to execute on a consecutive block of iterations. Therefore the total iteration count and block size must be determined at the entry of the loop. The loop boundaries that determine the exit of the loop must remain independent of the iteration number. Moreover, if there are `break` statements in the loop, the total iteration count may not be determined at the entry and therefore it is not applicable for DOALL block parallelisation.

To show how a DOALL loop is parallelised in the block approach, I give an example loop from the SPEC CPU2006 libquantum benchmark. The assembly version of the loop is shown in the left of Figure 5.10. Through the static binary analysis discussed in Section 5.2.2, register `rbp, rdi` and `r9` are identified as `ReadOnly` variables. Registers `rsi` and `r10` are modified in the loop but they are not live at entry of the loop, therefore they are `Private` variables. Registers `rax` and `r8` are identified as `Depending` variables since they are live at the entry of the loop and they are also defined in the loop body. Register `rax` is identified as the `Induction` variable as it is incremented with a constant offset `0x10`. And register `rdi` contains the check condition of the `Induction` variable.

When the execution reaches the tail of the `Init` block of the loop, the main thread saves its machine context to the virtualised shared structure. However, only recognised `ReadOnly` and `Depending` variables need to be spilled to the shared machine context. Once the main thread saves registers `rbp,rdi,r9,rax,r8`, it sets the valid flag so that all parallel threads can load the saved values into their respective private registers. After all registers are privatised, each thread calculates the total iteration count by subtracting the content of check boundary `rdi` and induction variable `rax`. Each thread then evaluates its initial value of `Induction` variable and the check boundaries according to the induction increment offset C, thread ID `ID` and the total number of thread T.

The whole piece of induction variable initialisation is JIT-compiled per thread with constant optimisations for each thread, which is marked as yellow in Figure 5.10. The resulting JIT code for induction variable initialisation can be summarised as follows:

$$\begin{aligned} \texttt{block} &= (((\texttt{rdi} - \texttt{rax})/C)/T) * C \\ \texttt{rax} &= \texttt{rax} + \texttt{block} * \texttt{ID} \\ \texttt{rdi} &= \texttt{rax} + \texttt{block} \end{aligned} \tag{5.2}$$

Once the induction variable is initialised, the `Depending` variable `r8` is also updated based on the fact that `r8` is depending on `rax`. After initialising of all depending variables and load all read only registers, the loop is set to run with updated boundaries. There are no other modifications in the loop body as shown in Figure 5.10 (left).

For a more generic handling of block based parallelisation, GABP generates specific initialisation routines based on the type of induction variables and check conditions that reside in registers, stack and memory locations. Privatisation is performed independently for each thread in the loop body, if there exists writes to a `Private` variable.

5.4.2 Cyclic Parallelisation

If the total iteration count cannot be dynamically determined at the entry of the loop, `DOALL_Block` based parallelisation is not applicable. Instead, GABP uses `DOALL_Cyclic` parallelisation, which schedules each thread to take turns and execute a single iteration in round-robin order. When a thread finishes the current iteration `i`, it continues to work on iteration `i+T`, where `T` is the total number of threads.

When one of the threads reaches an exit block, the last iteration can be determined. At this time, GABP needs to force other threads working on "future" iterations to abort. Since it is a parallel environment, threads may already be working on the iterations after the determined last iteration. In this case, their changes should be squashed and they should return to the thread pool immediately. Therefore it is necessary to buffer all memory writes performed by a thread, even though there are no cross-iteration dependencies. A `Commit` stage is added when a thread finishes the current iteration and flushes all the changes from its write buffer to the shared memory.

The `DOALL_Cyclic` loop requires the order of thread `commit` to be serialised. This is because the current thread can only commit its changes provided that the previous thread has confirmed it is not leaving the loop. If a thread leaves the loop early and reaches the loop exit block, it sets the `finish` flag so that other threads will know before committing their changes. Once the `finish` flag is set, all other threads flush their write buffers and immediately jump back to the thread pool.

The performance of `DOALL_Cyclic` parallelisation is considered worse than using the `DOALL_Block` parallelisation. Firstly `DOALL_Cyclic` requires all memory accesses to be buffered and an extra commit stage is required. Secondly, `DOALL_Cyclic` trashes the data locality that exist between adjacent iterations, which would result in worse performance on existing cache hierarchies and hardware prefetchers. Detailed overhead analysis is discussed in the Section 6.2.3.

5.5 Resolving Runtime Data Dependencies

Previous parallelisation solutions are only applicable on `DOALL` loops, where there are no cross-iteration dependencies except predictable induction and reduction operations. However, `DOALL` loops are rare in the general application spectrum. To extend GABP's applicability, we aim to parallelise irregular loops with complicated cross-iteration dependencies. The limit studies in chapter 4 show that it is feasible to achieve such parallelism on real hardware.

In GABP, there are two major components that address the cross-iteration dependencies: thread-level speculation and thread synchronisation. Compared to existing implementations from prior work, GABP maximises the efficiency using JIT-complied routines. For thread-level speculation, a JIT-friendly software transactional memory (STM) called `JIT_STM` is implemented. `JIT_STM` minimises its speculative operation costs by generating inlined code and integrating in the context of the dynamic binary translation.

5.5.1 Just-In-Time Software Transactional Memory

As GABP aims to achieve performance though thread level speculation on common multi-core processors, we are not expecting any assistance from special hardware. Therefore, I chose to focus on minimising overhead of *software transactional memory* (STM) through JIT compilation. Mis-speculation rates are also reduced with the help of value prediction guided by hint instructions from static analysis. Although there exist other STM libraries, such as TinySTM [131], they are external libraries which are not JIT-friendly and their runtime overhead is too large to bring performance during parallelisation as shown later in this section.

I propose `JIT_STM`, a word-based STM integrated in GVM, whose speculative operations are purely JIT-compiled and inlined in the original instruction stream. `JIT_STM` is a value-based STM with lazy conflict checking, which shares the most similarity with the JudoSTM [85]. As discussed in Section 2.3.5.2, the advantages of value-based STMs are that they are simple to implement and easy to generate by JIT compilation. The efficiency of `JIT_STM` can be further improved by the assistance provided by hint instructions.

5.5.1.1 Read and Write Buffer

`JIT_STM` achieves the atomicity of a transaction by sandboxing its read and write accesses in its read and write buffers. For all the read operations, only the initial load value and address are recorded in the read buffer. The read buffer is mainly used for conflict checking against the main memory to make sure the transaction's initial reads are not changed by other threads during the execution of the transaction.

The write-buffer not only buffers all the transaction's write accesses but also "silent reads" after write operations. Write isolation can prevent dependencies caused by write operations from other threads. Write values are only flushed to main memory or other threads after it validates all its initial reads of the transaction. If any of the recorded initial reads are different compared in the shared memory, all buffered writes of the transaction are squashed. It resumes to the start of the transaction and re-executes the transaction. If all the recorded initial reads have remained unchanged, then all buffered writes are safely flushed to the shared memory. Progress is always guaranteed by prioritising the execution of the oldest thread, the thread who executes the lowest iteration.

`JIT_STM` relies on the `SPEC_MEM_ACCESS` instruction to guide GVM to redirect dynamic memory accesses to its corresponding read and write buffers. Figure 5.11 illustrates the structure of the access redirection in `JIT_STM`. For each memory access that is marked as speculative by the hint program, the dynamic memory address is firstly obtained. Then we perform a table lookup with the address as the key to a redirection table (or privatisation map). If there is a hit in the redirection table, the redirected read or write buffer entry address is immediately returned and it can be directly referenced by the instruction implicitly. The code for the first query of the hash table lookup is inlined as shown in Figure 5.9, which is the same as privatisation memories.

The returned redirected address points to an existing entry from either the read or write set. If the runtime address hasn't been read before, an entry is created with the loaded initial value in its read set. The pointer to this entry is then recorded in the redirection map so that later accesses can directly refer to the read entry. If there is a write to this address, an entry of its write set is created by copying the value of the read set. The redirected address is switched to point to the write entry. All subsequent reads and writes are then redirected to the write entry, leaving the original read entry only recording the initial load value of the address.

5.5.1.2 Hint-Guided JIT Speculation

Even with the optimised structure for isolating transactions, the timing cost of hashing operations and read/write sets maintenance is still significant. If a runtime address is frequently read and written by many instructions during a transaction, a fixed write

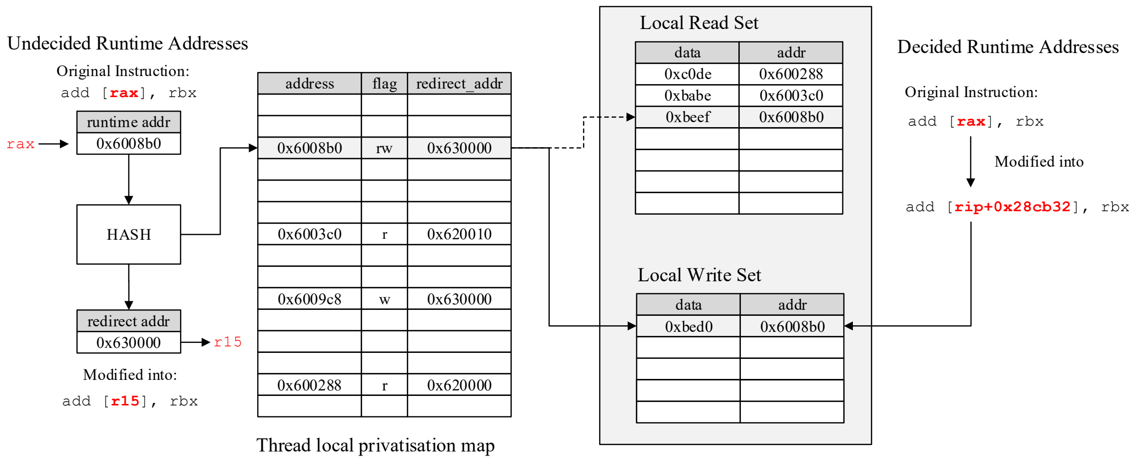


Figure 5.11: `JIT_STM` redirects a runtime address through hashing in a privatisation map (redirection table). The initial reads of the transaction are buffered in the read set and all subsequent accesses are redirected to the write buffer. If the instruction is statically proved to only access a dynamic address that is independent of iterations, the address of the corresponding write set entry can be JIT compiled into an immediate access, which avoids the need for hashing and redirection (on the right).

location can be assigned to the runtime address. All accesses from the instructions can be recompiled into direct memory accesses to this fixed location, which avoids the need for hash table for redirection. Similarly for handling speculative accesses to registers and shared stacks, they are also redirected to a fixed location based on the register ID and stack offset. As shown in Figure 5.9, the minimum required number of instruction for redirecting a memory access to read/write buffers is around six, while directly accessing the distinct write entry requires no additional cost.

Given a frequent dynamic memory address `addr` accessed by the set of instructions `access`(`addr`). It can be assigned with a fixed entry, for example, the `i`th in the local write buffer. Three conditions must be satisfied.

- **Thread-private code cache**: each thread JIT compiles and executes its own version of speculative code in its thread-private code cache, allowing different direct speculative accesses to be encoded as immediate accesses in its instructions. It is already provided in the GABP default infrastructure.

- **Fully independent accesses**: for instructions `e` and `i` where `e` $\notin$ `access`(`addr`) and `i` $\in$ `access`(`addr`) in the transaction, the dynamic accesses of `e` and `i` must not alias. The proof is performed by the static analysis and additional profiling is required to guarantee that no external pointers refer to runtime address `addr`.

- **Constant dynamic address**: for the instruction `i` where `i` $\in$ `access`(`addr`), `i` must not access other runtime address other than `addr` in the same transaction.

When all three conditions are satisfied and proved in static binary analysis, hint instructions are generated to annotate each instruction `i`, where `i` $\in$ `access`(`addr`) and deliver the information to `JIT_STM`. At runtime, the instruction memory access is modified into an absolute memory access to the allocated entry in the read or write buffers.

```
Original                   TinySTM (Pseudo code)     JIT-STM

addq [rbx+0x10], rcx       s0 <- spec_read(rbx+10)   addq [rip+0x289543], rcx
cmp  [rbx+0x10], rdx       s0 <- s0 + rcx            cmp  [rip+0x289546], rdx
jle  some_place            spec_write(rbx+10,s0)     jle  some_place
                           s0 <- spec_read(rbx+10)
                           cmp s0, rdx
                           jle  some_place
```

Figure 5.12: Assembly code generated to support speculation, using TinySTM and JIT-STM respectively

Therefore a speculative memory access can be optimised as a single memory access, and avoids the hashtable lookup.

Compared to other STM implementations where a single speculative memory access is expanded to tens of instructions, `JIT_STM` exploits the benefits of JIT compilation and achieves the same memory redirection by just using one instruction, which significantly reduces the operational overhead. The performance difference is illustrated in Figure 5.13. The overhead of `TinySTM` is measured by replacing a memory operation to an existing call to the `TinySTM` library. It is seen that `JIT_STM` achieves much faster than `TinySTM`. The results demonstrate that it is essential for me to implement `JIT_STM` to optimise for dynamic binary translation.

At binary level, it is difficult to integrate the `TinySTM` library calls in the original instruction stream. For example, Figure 5.12 illustrates the generated code to invoke two STMs. Since x86 is not a load and store architecture, memory operands might be accessed directly by operations such as the `add`. Unfortunately each single memory access is expanded to one or two procedure calls, making the speculative access very inefficient, let alone achieving performance through parallelisation. `JIT_STM` does not introduce new trampoline calls to STM libraries but assigns a direct dynamic thread-local address for the given dynamic address.

Besides the information for the elision of hash table lookup, all other operations in `JIT_STM` are also specified and controlled by hint program. The start and finish of a speculative transaction are annotated by two hint instructions `TX_START` and `TX_COMMIT`. Upon the start of the transaction, a thread saves its current machine context including its PC to a `checkpoint` buffer in its thread. Upon the commit of the transaction, the thread goes through the read validation buffer and validates the recorded value against the actual memory value from recorded address. The validation of registers and stack elements are performed between the thread's private version against the virtualised shared machine context shown in the Figure 5.8. After validation, it copies the private writes to the shared machine context in a critical section.

The hint program can also help JIT compilation by specifying free general purpose registers that are not used or least used during the transaction, so that they can be directly used as scratch registers with minimised usages of spill and restore operations. Once the free scratch register information is available, a free thread is recruited as *warden thread* to JIT-compile all the above `JIT_STM` routine code during the start of the application execution.

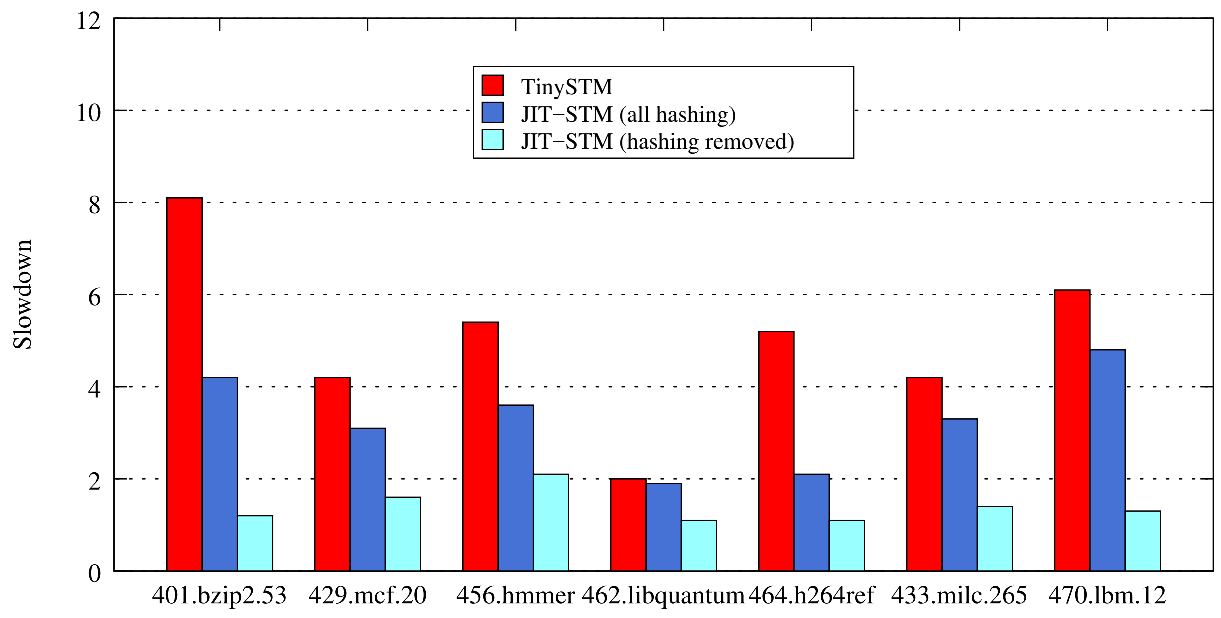


Figure 5.13: Single threaded overhead comparison between the implemented `JIT_STM` and the state-of-the-art STM `TinySTM 1.0.5` over a selection of speculative loops from the SPEC2006 benchmarks. The selected loop is surrounded in a transaction and all memory accesses are replaced as speculative accesses as shown in Figure 5.12. The slowdown is calculated by measuring the accumulated time of the speculative loop execution over the native sequential execution of the loop on Intel(R) Xeon(R) E5-2667.

5.5.1.3 Speculative Signal Handlers

When a thread is in a speculative state, it is very likely that many different segmentation faults or arithmetic faults would occur during the execution of the speculative code. This is because the input of the transaction speculates on a wrong input data.

`JIT_STM` implements two mechanisms to address the faults caused by mis-speculation.

- **Segmentation and other faults**: `JIT_STM` implements a signal handler that identifies the incoming signal and thread. If it is caused by the original application, then it passes the signal as normal. If it is a fault caused by mis-speculation within the transaction, the thread clears the remaining transaction, leaves the signal handler and rolls-back the execution to the valid `checkpoint` buffer. If frequent fault and aborts occur, the thread stays in a loop and waits until it is safe to rollback.

- **Random jumps caused by mis-speculation**: sometimes the target of an indirect branch or return address of the stack in the transaction might be wrong due to mis-speculation, the control flow will be directed to a random address without filing a signal. To address the problem, an additional check is performed in the `Transformation` stage of all the speculative basic blocks. If the target of a branch is observed to be outside the loop code, then it is mis-speculated and the transaction is dropped. The thread discards the translated code and performs a context switch to the valid `checkpoint` buffer.

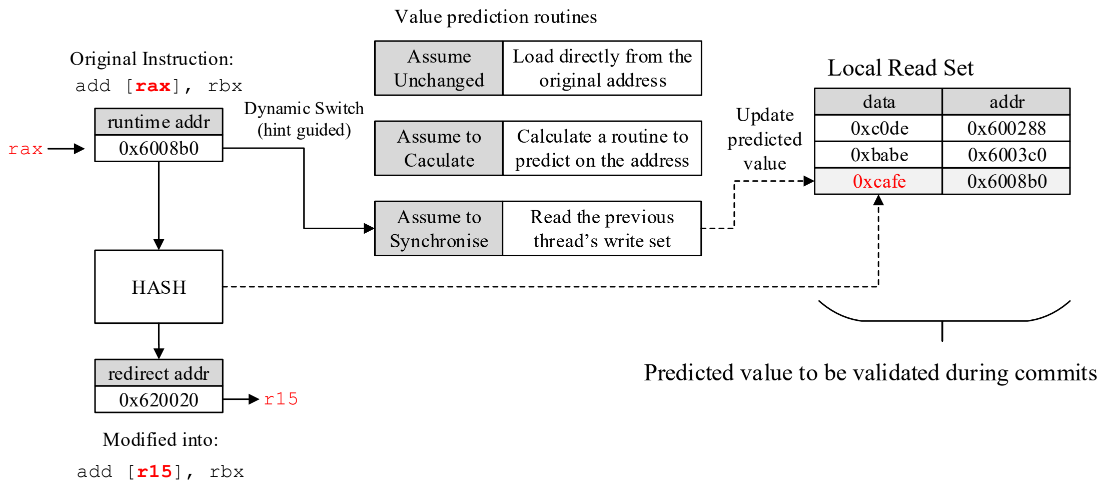


Figure 5.14: JIT_STM value prediction: three prediction policies can be selected by hint instructions. The predicted value is recorded in its read set.

5.5.2 Speculative Value Prediction

Even with minimised cost in the STM operations, the overhead caused by the nature of value-based STM is still significant. As discussed in chapter 4, the performance for thread-level speculation operations are highly sensitive to the transaction size. The overhead is even magnified by mis-speculation and re-execution. If mis-speculated, the whole read and write sets are flushed which is a waste of computing resources. In order to reduce the mis-speculation rate, one effective approach is to perform value prediction on the potential conflict reads of the transaction.

In JIT_STM, value predictions are performed in three different approaches:

- **Prediction 1**: the read value is predicted `ReadOnly` or `Unchanged`. This value prediction scheme is equivalent to thread-level speculation. When it performs a speculative read on the input address, the value stored on the memory address is directly loaded into the read set. It is similar to optimistically assuming that the address would not be modified by other threads.

- **Prediction 2**: the read value can be reproduced based on the fixed `update` pattern and initial value of the input. This assumption is used for the case when it is guaranteed that a data dependence would occur. This value prediction scheme is equivalent to optimisations on induction, reduction and other depending variables. The prediction of predictable variables has already been applied in the parallelisation of `DOALL` loops.

- **Prediction 3**: the read value can be fetched directly from synchronisation if the previous two assumptions always produce the wrong prediction.

GABP expands the scope for value prediction using all the three assumptions. In this section, I discuss this guided value prediction that selects the best value prediction schemes for a potential depending variable using hint instructions.

5.5.2.1 Guided Speculative Value Prediction

In GABP, the value prediction is only performed on `Depending` variables of a loop. The recognition of the `Depending` variable is discussed in Section 5.2.2. The value of a variable is predicted during the first read when creating a read entry in its local read set. Instead of loading from the shared memory, the value prediction can be performed at this exact time. The process is shown in Figure 5.14. For the first read of each `Depending` variable that is suspected to cause a violation, a hint instruction specifies the strategy for predicting this variable. Then the selected value prediction routine is inlined into the original instruction stream. After calculation, the predicted value is written to the corresponding entry in `JIT_STM` read set. The rest of subsequent reads and writes are handled exactly the same as `JIT_STM`.

Instead of assuming the load value to be unmodified, the second prediction scheme assumes the correct value can be reproduced by a calculation of a software routine: `update` function. The update function can be created from a subset of instructions that contribute to regenerate the read value. The update operation of the variable can be summarised as:

$$\mathtt{var} = \mathtt{update}(\mathtt{var}, \mathtt{tid}, \mathtt{C}_1, \mathtt{C}_2, ...\mathtt{C}_\mathtt{N}) \tag{5.3}$$

where `tid` is the thread ID and $\mathtt{C}_1$, $\mathtt{C}_2$ ... $\mathtt{C}_\mathtt{N}$ are all constants or read only variables in the scope of the loop. If the *update* function is a closed and simple form, the cascaded form of update function can be encoded into a small code snippet. The snippet can be inlined during value prediction, so that the value for any given loop iteration can be regenerated.

5.5.2.2 Speculation or Synchronisation

If the *update* function could not be transformed into a closed form or the update function is too sophisticated to be JIT compiled, then the second method for prediction is not applicable. Instead of performing a prediction that is very likely to be wrong, the third method is chosen that waits until the depending data has been computed by the previous thread. This is particularly beneficial when parallelising big loops with very frequent dependence pairs with high rollback costs. For frequently executed dependence pairs, synchronisation is a better option than speculation since the cost of transaction aborts and re-execution is much larger than the cost of transmitting data. For rare dependence pairs, speculation is better since for the most of the time, the high cost of roll-back costs is not incurred.

We define P_{ij} to be the probability of average data dependence violation that may occur between the read of instruction i and the write of instruction j from the other thread. Assume the cost for synchronisation is C_{Sync}, the cost for TLS without rollback is C_{TLS} and the cost for re-execution is $C_{rollback}$.

$$C_{Sync} = (1 - P_{ij}) * C_{TLS} + P_{ij} * C_{rollback} \tag{5.4}$$

We define a threshold probability P_T such that

$$P_T = \frac{C_{Sync} - C_{TLS}}{C_{rollback} - C_{TLS}} \tag{5.5}$$

If the probability of data dependence violations that is greater than P_T, then we prefer the first read of the `Depending` variable should wait for the previous thread's writes.

While for a read that has the probability of violation less than P_T, assuming the read data won't change would bring more benefits. As for implementation, the probability of data dependence pairs is obtained through dynamic profiling. The profiled probability can be obtained by counting the number of total violations caused by the data dependence over the total iteration number.

A mixture of synchronisation and speculation in one transaction would have an effect of weakening the atomicity of transactions. Firstly it can't guarantee that there are no conflicts between transactional and non-transactional accesses. Secondly, data read from synchronisations within a transaction are also speculative results sent from previous thread. This data is unreliable and it is also needed for validation before commits. To guarantee strong atomicity, it privatises all writes to registers, stacks and memory locations within the iteration. If the data read from the previous thread results are in violation, the current transaction should also abort, which creates a chain of mis-speculation.

5.5.2.3 Versioned Signal and Wait

The implementation of speculative synchronisation uses the same local read and write buffer from `JIT_STM`. Synchronisation is only invoked by the specification of hint instructions. Forwarding data between threads is split into two operations: `Signal` and `Wait`, which is the same as `DoAcross` parallelisation. The `Signal` procedure is inlined when the last write of the depending data has been updated, while the `Wait` operation is inserted before the first read of the depending variable. Specifically, it is inlined in the value prediction procedure during the creation of an entry in the local read set as shown in Figure 5.14. As all the writes are privatised, it does not need to consider `WAR` and `WAW` dependencies.

Algorithm 2 shows the details of the `Signal` and `Wait` operation. Compared to the synchronisation scheme from `HELIX`, our approach uses versioned and non-blocking signals in the speculative context. If a signal is sent speculatively, it must process the ability to undo the effect of the signal in case the transaction has been aborted. By sending the version of the variable data, the next thread is able to differentiate the valid version of data it could read. In the `Wait` operation, the thread firstly checks whether it is the "oldest", which means all previous threads have commited the changes for previous iterations. Then it can safely load from the shared memory. Otherwise if the thread is not the oldest, it checks whether the version of the variable matches its current iteration. If the version of the depending variable is less than the current iteration number, the thread spins and waits until the version is updated. Once the variable value is loaded, it is copied to the corresponding entry of the read validation buffer for verification before its commits.

Making the `Signal` non-blocking enables signals to be resent in case of a transaction abort. A thread may resend the `Signal` with updated value of the variable without waiting for the next thread to consume the data. Even with incorrect data sent from synchronisation, threads can recover from incorrect execution since the value from the `Wait` operation is treated as a predicted value. Hence the transmitted value would be verified before the commit operation of a thread.

Algorithm 2 Signal and Wait operations for speculative synchronisation

```
Ensure: $\forall v \in \mathbf{V}$ channel[v.ID] is allocated.
 1: procedure SIGNAL($v$)                ▷ Signal the validity of variable $v$ to next thread
 2:     channel[v.ID].data = v
 3:     channel[v.ID].version = thread.iteration + 1
 4: end procedure
 5: procedure WAIT($v$)         ▷ Wait until the variable $v$ is valid from previous thread
 6:     if thread.oldest then
 7:         v = $v$ in shared memory
 8:     else
 9:         while channel[v.ID].version < thread.iteration do
10:             wait
11:         end while
12:         v = channel[v.ID].data
13:     end if
14:     validateReads.insert(v);
15: end procedure
```

5.6 Generic Loop Parallelisation

We discussed the implementation to handle runtime dependencies using thread level speculation. Compared to other approaches, the JIT_STM exploits JIT capabilities to reduce memory redirection overheads and uses guided speculative value prediction to reduce mis-speculation rates. The final stage is to perform parallelisation on generic Non-DOALL loops with cross-iteration dependencies. Parallelisation is the combination of all techniques discussed in this chapter. These techniques are selectively enabled based on the hint program from static binary analysis.

To demonstrate the process of how a generic loop is parallelised, consider the following loop taken from the integer SPEC CPU2006 benchmark `473.astar` from its function `wayobj::makebound2`. Since it is not helpful to present the original binary assembly, the source code is listed here.

```
int bound2l=0;
for (i=0; i<bound1l; i++) {
  index=bound1p[i];
  index1=index-yoffset-1;
  if (waymap[index1].fillnum!=fillnum)
    if (maparp[index1]==0) {
      bound2p[bound2l]=index1;
      bound2l++;
      waymap[index1].fillnum=fillnum;
      waymap[index1].num=step;
      if (index1==endindex) {
        flend=true;
        return bound2l;
      }
    }
  index1=index-yoffset;
  if (waymap[index1].fillnum!=fillnum)
    if (maparp[index1]==0) {
      bound2p[bound2l]=index1;
20       bound2l++;
21       waymap[index1].fillnum=fillnum;
22       waymap[index1].num=step;
23       if (index1==endindex) {
24         flend=true;
25         return bound2l;
26       }
27     }
28   ...
29 }
```

The loop contains many early exits in the loop body, where the iteration number could not be determined at the start of the loop. Therefore the cyclic DOACROSS parallelisation approach is only applicable. Statically, it is also impossible to determine whether the loop contains cross-iteration dependencies due to data-dependent indirection of array indexes `index=bound1p[i]` in line 3 and 4. Therefore, we rely on the information from profiling the program with training inputs. From profiling through BEEP from all training inputs, the dependence patterns converge, which means the loop has a fixed dependence pattern regardless of inputs.

The following shows the BEEP parallelism report:

```
Profiled true dependence pairs:
Line 5 and 21 on waymap[index1]: Probability 1.5%
Line 17 and 9 on waymap[index1]: Probability 0.8%
Line 7 and 20 on bound2l : Probability 9.8%
Line 19 and 21 on bound2l: Probability 5.7%
```

From the probability of each dependence pair, the static binary analyser determines an optimal scheme to resolve dependencies. It is observed that all the four dependence pairs occur with rather low frequency. Therefore it is beneficial to use thread-level speculation to handle runtime dependencies due to the estimated low mis-speculation penalty. The static binary analyser marks the loop as a `DOACROSS_SPEC` loop and generates hint instructions to guide speculative execution. Two hint instructions `TX_START` and `TX_COMMIT` are annotated at the beginning and finish of the loop iteration code, so that the whole iteration is surrounded in a transaction. The accesses to `waymap[index1]` and `bound2l` are marked with `SPEC_READ` and `SPEC_WRITE` hint instructions. During runtime, their dynamic accesses are redirected to read and write buffers. All other memory write accesses must also be annotated with `SPEC_WRITE` to isolate the changes within the transaction. The rest of hint generation remains the same as `DOALL_CYCLIC` parallelisation.

5.6.1 Hint Generation Strategy

Determining the best value prediction of each `Depending` variable is the key to reducing overall mis-speculation rates and achieve performance. The decision is made by the static binary analyser delivered by hint instructions. The `Depending` variables in this example are `i`, `waymap[index1]` and `bound2l`. `i` can be identified in static analysis as an induction variable. Its value can be regenerated based on the iteration number and its update offset. For the rest of the variables, it is found in the profiling dependence report that both variables have relative low conflict rates, therefore it is better to use the first prediction scheme that assumes the variable remains unchanged.

In conclusion, the strategy for generating hints to safely resolve potential dependencies among all different accesses in a transaction is given as follows:

- For all instructions that access variables that need synchronisation, redirect both reads and writes. They are subject to speculative validation and commit.
- For all instructions that access speculative data, redirect both reads and writes. They are subject to speculative validation and commit.
- For all instructions that don't access synchronised nor speculative data, buffer all their writes. They are not subject to speculative validation but need to commit.
- For all the other instructions where it is not certain whether they are aliased with synchronised data or not, buffer all their reads and writes. They are subjective to speculative validation and commit.

5.6.2 Correctness and Verification

5.6.2.1 Static Consistency Verification

The philosophy behind GABP is to decompose the whole automatic parallelisation transformation into two layers. The first, encapsulated as a hint program, controls the consistency and order of high-level coarse-grained transformations. The second, specified by each hint instruction, corresponds to the individual transformations performed by each GVM modification handler.

In this manner, GABP maintains correctness and consistency by guaranteeing that all GVM handlers combined maintain program semantics after modification. If a GVM handler makes changes that alter the behaviour of a basic block then the hint program must contain other handlers that restore it to the original state. For example, if a hint instruction is used for switching to another user-defined stack. GABP must ensure that there is another hint instruction later on in the hint program that switches the stack back to its original. As the correctness of each individual handler can be easily checked, verification is reduced to ensuring the consistency of the hint program through definition of each hint instruction's semantics.

Hint program generation must be safe and reliable, because a tiny error would result in incorrect translation of the original application. To maintain consistency between static and dynamic components, the static analyser must include an accurate model of GVM with the same assumptions on handling program structures (basic blocks, loops, etc) as the dynamic binary recompilation engine, so that the transformations specified by the static analyser are carried out as required.

5.6.2.2 Dynamic Runtime Validation

GABP also has a series of dynamic checks to ensure the correctness of parallelisation. Firstly, the program output under GABP must be exactly the same as the output from the native executable. During execution if a segmentation fault or other signal occurs, GABP performs address lookup to determine whether the signal was intended for the original application or was generated due to GABP modification.

Secondly, GABP uses software transactional memory to buffer speculative accesses and value prediction data in a transaction. Runtime validation is performed to detect any potential data violations that occurred during the parallelisation. Correctness is also enforced by rolling back to a sequential execution if a data violation is detected. However this check must be explicitly controlled by the hint program

Thirdly, some applications contain self-correctness checks and assertions embedded in their executable. These self-checks can also be an effective scheme to validate the modification performed by GABP.

5.7 Related Work

The principle of binary parallelisation in GABP is not to invent new parallelisation methods but adopt existing techniques from compiler-based automatic parallelisation techniques and implemented in an efficient way at a machine code level. Due to the existing difficulties in binary translation and parallelisation, I'm not aware that there was much work that focuses on binary parallelisation.

Aparna Kotha et al. [132] proposed a pure static approach that parallelises affine loops and rewrites the binary in the `SecondWrite` binary rewriter. They achieved substantial performance on regular benchmarks from Polybench. However, there exist a few limitations on their work. Firstly their scope of parallelisable loop is restricted—only affine loops with linear array accesses and fixed loop boundaries can be parallelised. For generic executables with irregular loops, the coverage of affine loops is typically low. Secondly, they parallelise the binaries by rewriting them statically, which is problematic for binaries without relocation information. Although they claim that their `SecondWrite` binary rewriter is able to convert whole binaries to LLVM IR and optimise them, the resulting translation is typically speculative, conservative and not always applicable for all optimisation passes. To maintain the correctness of static binary translations, runtime checks are added and the original executable is appended as a fallback path, in case a problem exists. While our tool is only able to convert a fraction of the executable into LLVM IR, the optimised code can be linked back during runtime seamlessly. A comparison of performance will be discussed in the next chapter.

Efe Yardımcı and Michael Franz [133] presented a framework using a combination of static and dynamic binary parallelisation slightly similar to our approach. They propose a software layer to be added as a dynamic binary translator. However they only target DOALL loops and parallelise them in block parallelisation on a PowerPC machine. Their parallelisation performance is not substantial and therefore they perform further vectorisation to improve performance.

The above two studies are the only work I am aware of that demonstrate real performance on real machines. There exists other research that evaluates speculative parallelism through simulation and limit studies. This is due to the fact that thread-level speculation is not supported with any existing commercial hardware.

Hertzberg et al. [134] proposed a runtime automatic speculation parallelisation framework called `RASP`. RASP encapsulates many optimisations similar to GABP such as thread-level speculation, value prediction, induction/reduction optimisations, synchronisation and loop unrolling. However the support of TLS is simulated with over-simplified assumptions. They do not handle many corner cases such as signal handling and error recovery.

Jing Yang [135] proposed a dynamic binary parallelisation system based on TLS. In the system, the main thread identifies hot traces from runtime execution and schedules workloads to other working threads for parallelisation. All the runtime data dependencies are handled by speculative execution from working threads. However, Yang's approach is only profitable for very long parallel traces (loop body) in order to outweigh the speculation overheads. His work was also based measurements from simulation with simplified assumptions on the overhead of speculation.

In conclusion, there are still no generic binary parallelisation frameworks that target general-purpose applications and run efficiently on commodity processors like x86-64 platforms.

5.8 Summary

This chapter describes the implementation of the GABP automatic binary paralleliser. It is an extension of GBR that enables automatic parallelisation decomposition in static analysis and recompilation in GVM to automatically extract thread-level parallelism on-the-fly, under the direction of its generated hint program.
The GABP implementation can be divided into static and dynamic GBR components:

- Static binary analysis: loop recognition, dependence analysis, alias analysis, loop characterisation, loop selection and how hint programs are generated from the retrieved information.
- Dynamic binary translation:
  - Thread management using thread state FSM.
  - Thread privatisation where register, stack and heap accesses are isolated through JIT optimisation on thread-local storages.
  - Runtime dependence detection and enforcement: `JIT_STM` and guided speculative value prediction.

In the next chapter, the performance of the proposed GABP framework is evaluated.

Chapter 6

System Evaluation

In this chapter, the actual performance of the proposed automatic binary paralleliser GABP is evaluated. Following on from the results of the parallelism study in chapter 4, a selection of loops in the SPEC CPU2006 benchmark are used for parallelisation under GABP. These loops are estimated to achieve overall program performance using BEEP's ideal and realistic parallel execution models. This chapter evaluates GABP performance on two different hardware platforms representing server and desktop environments.

6.1 Experimental Setup

The SPEC CPU2006 executables are compiled by a third party on a `x86_64` machine. This is to represent the case of legacy executable where we don't have the control of compilation nor the access to source code. According to the third party, the executables are compiled by `gcc` with `−O3` optimisations without vectorisation extensions.

The first machine I use is an Intel(R) Xeon(R) E5-2667 v4 processor which represents a server-class CPU. The second machine represents a desktop-class machine with Intel(R) i7 3770K processor. Details of the machines are summarised in Table 6.1. Frequency scaling (turbo boost) is disabled for both machines. The CPU affinity is configured to bind each GABP parallelsing thread to a hardware CPU core. The same set of the "legacy" executables are copied to the two mentioned machines for parallelisation. As the

Machine	Intel(R) Xeon(R) E5-2667 v4	Intel(R) Core(TM) i7-3770K
Total Cores	16	4
Cores Per Socket	8	4
Threads Per Core	2	2
Frequency	3.1GHz	3.4 GHz
L1 ICache	32K	32K
L1 DCache	32K	32K
L2 Cache	256K	256K
L3 Cache	25600K	8192K
DRAM Size	256G	16G
DRAM Channels	4 (per socket)	2
DRAM Bandwidth	76.8 GB/s	25.6 GB/s
OS	Linux Ubuntu 16.04.3 LTS	Linux Ubuntu 16.04.3 LTS
Kernel Version	4.4.0-112-generic	4.4.0-112-generic

Table 6.1: Specification of the two machines used for evaluating GABP performance.

Benchmark	ID	Contained Function	Coverage	Loop Type	Compiled Language
401.bzip2	32	`mainSort`	35.6%	`Speculation`	C
401.bzip2	68	`mainSort`	2.3%	`DOALL_BLOCK`	C
410.bwaves	1	`mat_times_vec_`	28.2%	`DOALL_BLOCK`	Fortran
410.bwaves	1	`jacobian_`	60.1%	`DOALL_BLOCK`	Fortran
429.mcf	23	`price_out_impl`	27.4%	`Synchronisation`	C
431.milc	265	`add_force_to_mom`	22.7%	`DOALL_BLOCK`	C
435.gromacs	83	`inl1130_`	74.2%	`DOALL_BLOCK`∗	Fortran&C
436.cactusADM	191	`staggeredleapfrog2_`	89.2%	`DOALL_BLOCK`	Fortran
456.hmmer	112	`P7Viterbi`	85.6%	`DOALL_BLOCK`∗	C
459.GemsFDTD	965	`updateh_homo`	26.7%	`DOALL_BLOCK`	Fortran
459.GemsFDTD	968	`updatee_homo`	25.8%	`DOALL_BLOCK`	Fortran
462.libquantum	15	`quantum_cnot`	13.0%	`DOALL_BLOCK`	C
462.libquantum	16	`quantum_toffoli`	61.7%	`DOALL_BLOCK`	C
462.libquantum	20	`quantum_sigma_x`	13.8%	`DOALL_BLOCK`	C
464.h264ref	890	`SetupFastFullPelSearch`	48.1%	`DOALL_BLOCK`	C
470.lbm	12	`LBM_performStreamCollide`	89.2%	`DOALL_BLOCK`	C
473.astar	113	`ZN6wayobj10makebound2EP`	21.3%	`Speculation`	C++
482.sphinx	17	`approx_cont_mgau`	36.2%	`Speculation`	C
482.sphinx	554	`vector_gautbl_eval_logs3`	37.2%	`DOALL_BLOCK`	C

Table 6.2: Details of the selected loops for evaluating parallel performance in Figure 6.1. Loop types that are marked with ∗ refer to manual assistance in hint generation to remove cross-iteration dependencies.

hypothesis in this dissertation focuses on the feasibility of parallelisation, the performance variation when altering NUMA topology is beyond the scope of our discussion. Therefore, for the Xeon server processor, I only use one of the two sockets for performance evaluation, although the performance may be implicitly impacted from the other socket.

We evaluate the performance of parallel execution by measuring the total elapsed time to execute the benchmarks recompiled through GABP. The elapsed time is measured by the Unix `time` utility, which represents the waiting time experienced by users. The inputs of the SPEC CPU2006 benchmark are the `reference` inputs, so that programs run for the longest time available using inputs from the benchmark suite. The output of the parallelised benchmark is also validated with the original sequential application output to ensure the correctness of the parallelisation. Given the multi-threaded and non-deterministic nature of the parallelised programs on a real system, each experiment is repeated for a minimum of 10 times. We record the median, maximum and minimum elapsed time for the 10 runs.

Table 6.2 shows the details of loops selected for parallelisation and evaluation. These loops were chosen from BEEP profiling results shown in Table 4.3 that are proved to contain enough parallelism to achieve speedup. Most of the loops are `DOALL` loops or loops that can be transformed into `DOALL` loops after induction/reduction optimisations or removing other predictable cross-iteration dependencies. The set also contains several loop examples for thread-level speculation and synchronisation.

Note that there are loops in BEEP profiling results that are not selected. This is because they require significant engineering and implementation efforts in GABP recompilation to achieve correct parallel execution in a real system. As GABP is still in its prototype stage, there are a few corner cases that have not been addressed. I leave the rest of the implementation to the future and focus on the benchmarks with moderate implementation effort.

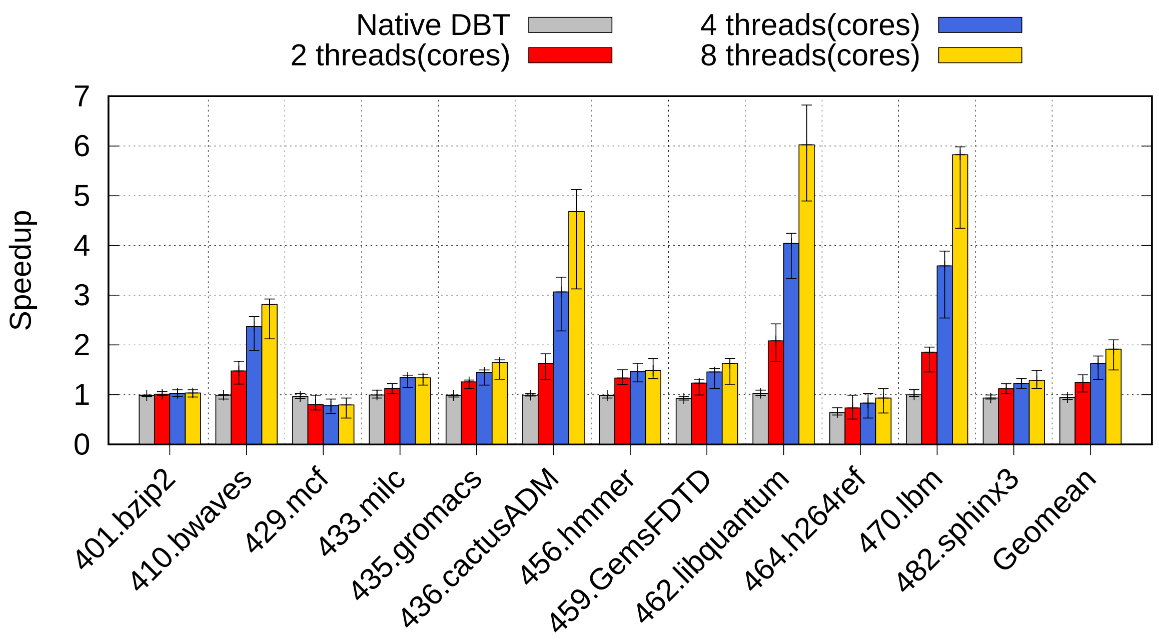


Figure 6.1: Whole-program speedups of benchmarks from SPEC CPU2006 achieved by GABP, using two, four, or eight threads (each thread is bound to one core) on the Intel(R) Xeon(R) E5-2667 processor. Native DBT refers to the native execution time under dynamic binary translation from DynamoRIO on a single core.

6.2 Performance Evaluation

Figure 6.1 shows the measured speedups of whole application runs on the Xeon processor. The baseline of the comparison, is the total execution time for running the original executable natively with reference input measured by the `unix time` utility. The time for parallel execution includes the total recompilation and execution time of the same executable under GABP using the same reference input. Figure 6.1 shows that GABP is able to achieve substantial performance through parallelisation on real systems, despite the overhead of dynamic binary translation. The geometric mean of the speedups on our example eight core CPU is 1.91×, with a maximum of 6.06×.

Benchmarks `462.libquantum` and `470.lbm` are two representative integer and floating point C benchmark programs that exhibit `DOALL` parallelism. Substantial speedup can be achieved due to the coverage of `DOALL` loops over the whole program execution. Similarly benchmarks `410.bwaves` and `459.cactusADM` represent executables compiled from the scientific domain written in the `Fortran` language. `401.bzip2` and `429.mcf` represent generic integer benchmarks that exhibit limited parallelism. The current implementation of GABP fails to achieve performance gain through parallelisation on this type of application, despite the usage of JIT runtime dependence handling such thread-level speculation. More detailed analysis is discussed in Section 6.3.

Figure 6.2 shows the ratio of the generated hint program size over the original executable size. It is clear that the hint programs are generally small (less than 10%), although they can reach over 10% if there are many transformations to apply. The size of each hint program reflects the degree of extra information needed to enable parallelisation at runtime.

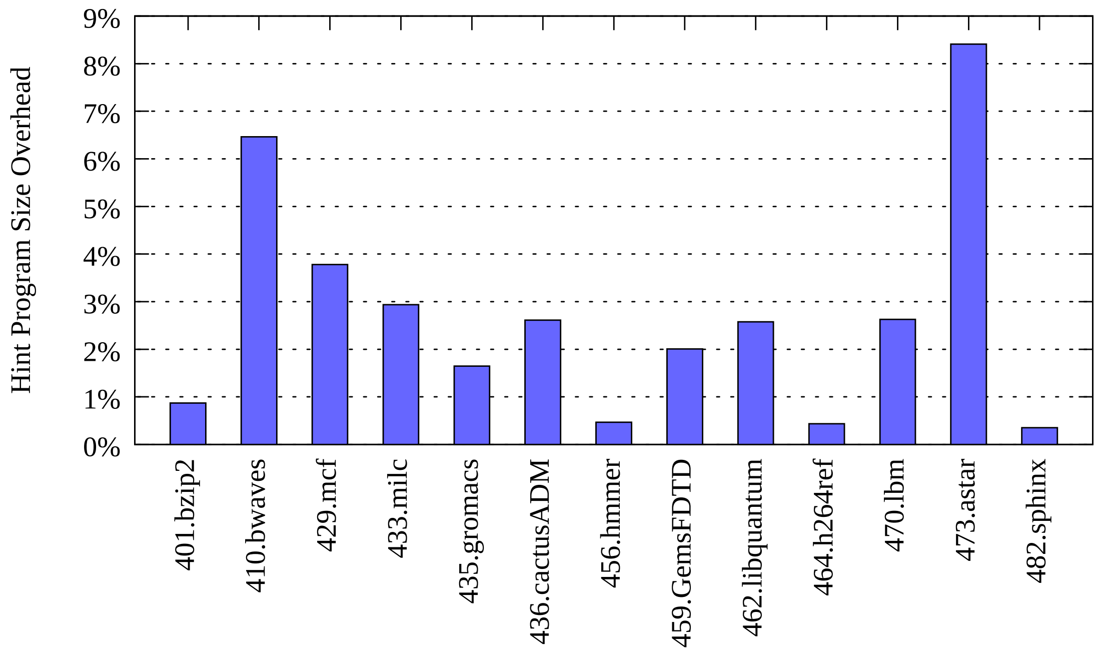


Figure 6.2: Ratio of hint program size normalised to the size of each SPEC2006 benchmark executable to guide automatic parallelisation.

As shown in Figure 6.1, the parallel performance can be achieved from benchmarks compiled from different languages, it demonstrates that GABP is source language agnostic. For example, `435.gromacs` denotes the type of applications that are compiled from a mixture of two languages `C` and `Fortran`. If there is no front-end support for a given language, it is typically impossible for conventional automatic parallelisation techniques to perform transformations.

However in practice, the difficulty in retrieving parallelism through binary analysis varies depending on the original language and compiler optimisation level. `Fortran` binaries are much easier to analyse than `C` executables since they typically exhibit regular control flow and affine memory accesses. For `C++` benchmarks compiled from object oriented paradigms, data-flow analysis is typically difficult as most data-flow is performed through class elements implemented in heap. Virtual calls are also frequently used and implemented as indirect calls, making it difficult to decompose parallelism and perform transformation.

6.2.1 Overhead Analysis

To measure the time breakdown of real execution for runtime overhead analysis, I use DynamoRIO's profiling `prof_pcs` runtime feature. `prof_pcs` issues a Linux timer signal that interrupts the underlying application code at a fixed frequency. It samples the `PC` of the interrupts and looks up the head `PC` of its containing basic block or trace. The trace head addresses are then recorded into a log file. By post-processing the log file, the time breakdown of different thread and application stages can be calculated. Note that all measurements are performed on the reference input of the SPEC 2006 benchmarks, where the average sequential execution time is around five to ten minutes. The execution time is long enough to overcome the probing effect caused by the interrupts at a moderate sampling frequency at 1 milliseconds.

To measure the wait time taken in the thread FSM spin locks, two specific hint instructions are introduced: `RDTSC_START` and `RDTSC_END`. The hint instructions instruct GABP to insert a `rdtsc` instruction that reads the `x86` time-stamp counter and buffers it in thread-local storage. When the lock is acquired, another `rdtsc` instruction is inserted so that the waiting time can be measured. Different counters are JIT inlined in the respective critical boundaries of thread state transitions, so that the probing effect is minimised. The measured overhead should reflect the overheads without the timer code inserted.

We firstly measure the time breakdown for benchmarks whose loops are parallelised with the `DOALL_BLOCK` type only. The rest of the loops using other parallelisation approaches are discussed in Section 6.3. The parallel execution time for `DOALL` loops can be attributed to six major sources:

- Translation overhead: the time for GVM to interpret hint instructions, perform translation and copy the translated code to a code cache.
- Indirect branch lookup overhead: the time for DynamoRIO to find the next translated block if the current block ends with an indirect branch. It is a major overhead for dynamic binary translation.
- Parallel fraction: the fraction of time for parallel execution. The parallel fraction can be recognised if the PC belongs to the loop body or its sub-functions of the selected loops.
- Sequential fraction: the fraction of time to execute other parts of the executable sequentially. It is the part that is not optimised.
- Loop init/finish overhead: the time for threads to perform induction variable initialisation and merge of private copies of reduction variables. It also includes the time to perform context switches between the thread pool and actual loop code during the start and finish of a loop.
- Spin lock overhead: the waiting time in spin locks for GABP to control threads in the thread state FSM. For cylic based parallelisation, it also includes the wait time to perform cyclic commits.

The sampled PCs are collected and sorted into the six buckets. By counting the number of trace heads for each bucket, the approximate weight of the execution time breakdown is calculated. Figure 6.3 shows the breakdown of the execution time measured on the E5-2667 machine. We can see that the overhead caused by dynamic binary translation is negligible as the translation occurs only once and most of the time is spent in the translated code in the code cache. For benchmark `464.h264ref`, the overhead for handling indirect branches takes around 18% for the single thread execution time. The overhead is due to the existence of indirect branches or altering of function pointers in the hot loop. Looking at their original source code, it can be seen there is a function pointer that gets assigned and called in its hottest loop in the function `SetupFastFullPelSearch`. The indirect call results in DynamoRIO's inlined indirect branch lookup for each iteration, which incurs significant overhead. For other benchmarks the indirect branch lookup overhead is negligible.

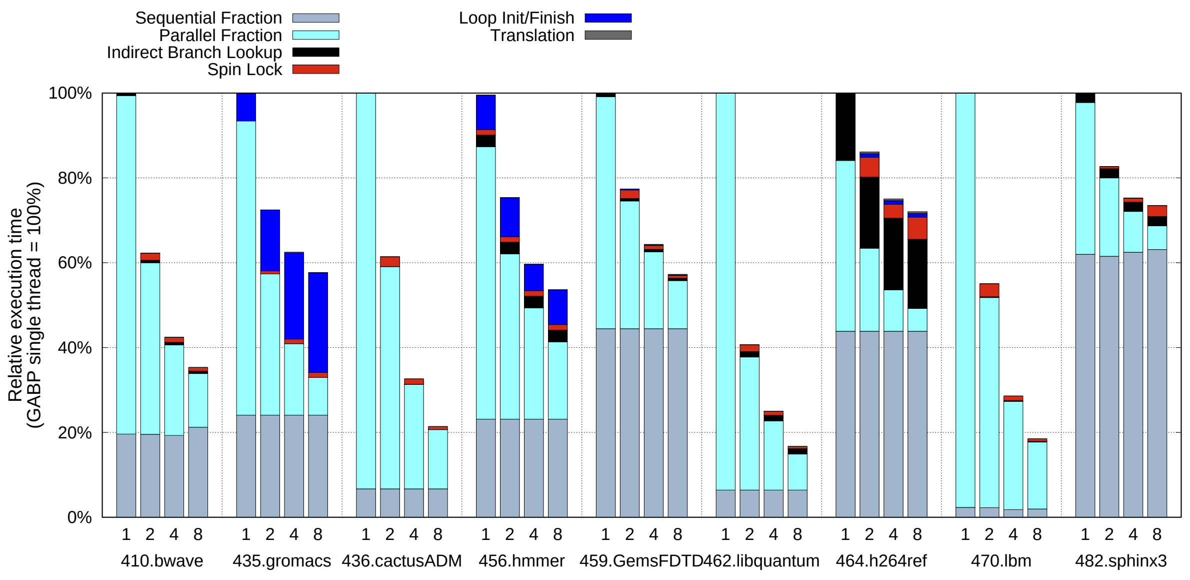


Figure 6.3: Breakdown of execution time for benchmark with `DOALL_BLOCK` loops on 1, 2, 4 and 8 threads on Intel(R) Xeon(R) E5-2667.

In terms of parallel execution, the time spent in the parallel fraction shrinks linearly as the number of threads increases. The parallel execution therefore follows Amdahl's law. Besides the execution time for the parallel execution, the overhead for loop initialisation and merge is also negligible. As for `DOALL_BLOCK` parallelisation, the loop initialisation and merge occurs only during the start and finish of the loop, which takes a tiny fraction of the total execution time as long as the time to execute the loop body is long enough.

However, there are exceptions for benchmarks `435.gromacs` and `456.hmmer`. All cross-iteration dependencies in `435.gromacs` exhibit reduction operations on different array elements. Therefore, dependencies can be removed by privatising the whole array data structure for each thread, turning the selected loop into `DOALL_BLOCK` form. During the loop finish stage, GABP merges all thread-private copies of the array into the original array. Therefore for `435.gromacs`, a large fraction of time is spent in the loop merging operation. The current merge time scales as the number of threads increases. However it can be optimised by hierarchically parallelising the merge operations though this complicates the thread controls. For `456.hmmer`, the cross-iteration dependencies are removed by splitting the original loop into a `DOALL` loop and a serial loop. The part executed sequentially is JIT compiled into the loop merge stage. Therefore we also see a large fraction of time spent in the loop finish stage due to the serial loop being inlined in the merge operations. Moreover, the parallelisable loop in `456.hmmer` is an inner-most loop, which is invoked millions of times by the outer loop. The overhead of the loop initialisation and merge process is magnified by the high invocation count of the inner loop. It can be reduced by having threads reuse initialisation resources and avoid register duplication across loop-invocations.

6.2.2 Machine and System Variance

For the second machine, with the desktop-class i7-3770K CPU, GABP shows a different performance result (see Figure 6.4). On i7-3770K, the parallel performance is much less

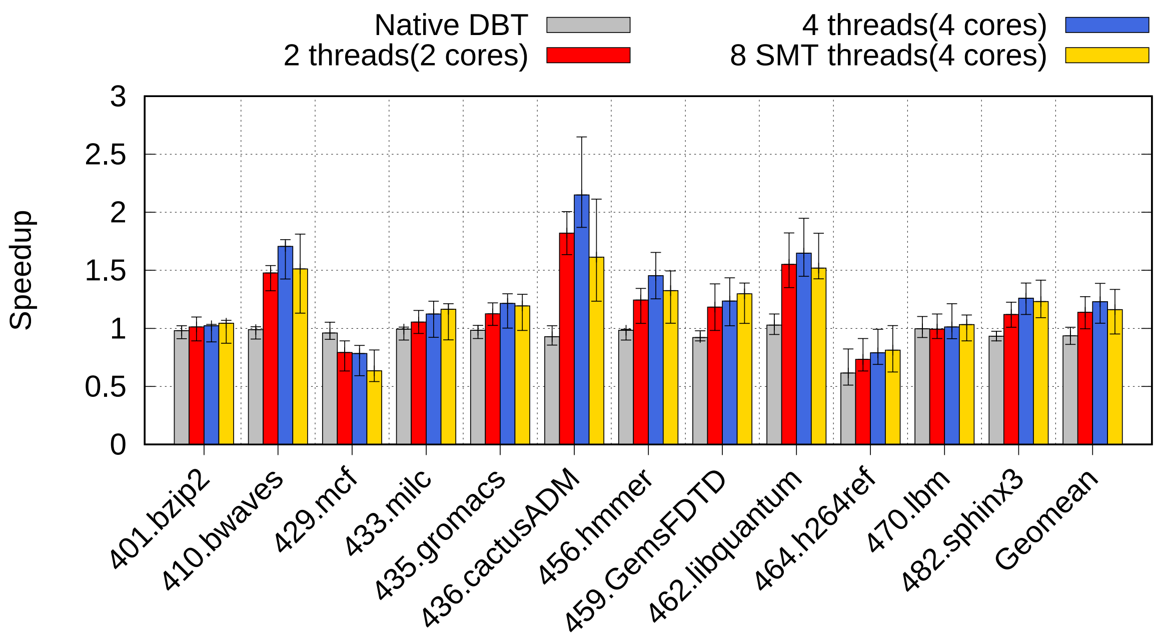


Figure 6.4: Whole-program speedups of benchmarks from the SPEC CPU2006 achieved by GABP, using two, four, or eight threads on the Intel(R) Core(TM) i7-3770K processor.

compared to the server-class CPU. Benchmarks such as `401.bzip2` and `429.mcf` do not see any performance improvement. Even for benchmarks with a high proportion of parallelism, such as `470.lbm`, see a massive drop in performance compared with the server CPU.

For a real system, there are many factors that impact the final performance of the parallelisation. Compared to simulation-based evaluation, it is not easy to quantify the exact source of factors that influence the parallelisation performance. Using the same overhead sampling technique, the execution time breakdown is obtained, shown in Figure 6.5. The overhead for translation, indirect branch, loop init/merge and spin locks does not vary significantly compared to the Xeon E5-2667 machine. However, the time spent in parallel fraction no longer correlates to linear scaling.

To further understand the reasons behind the performance difference in the parallel fraction, both software and hardware aspects are investigated. In terms of the software side, the subsequent modified code in GABP observed on i7 is the same as the code on Xeon. Moreover for programs with `DOALL` parallelism, there are no cross-iteration dependencies for communication between cores nor other modifications involved within the parallel fraction of the code. Given this, the performance difference could be due to the operating system schedulers. On the Xeon server, with two sockets, while GABP will run on one socket, other processes will run on the other and may interfere with GABP when both are accessing the memory system. For the small i7 desktop system, with just four cores, any benchmark parallelism above three threads is likely to be compromised by other user processes, especially the X11 server processes. At least one thread could be descheduled to run OS tasks. The overall performance is very sensitive to the OS descheduling since all the other threads have to wait for the descheduled thread to proceed to the next loop.

Both the i7 and Xeon CPU have the same L1 and L2 cache sizes. Figure 6.5 shows that

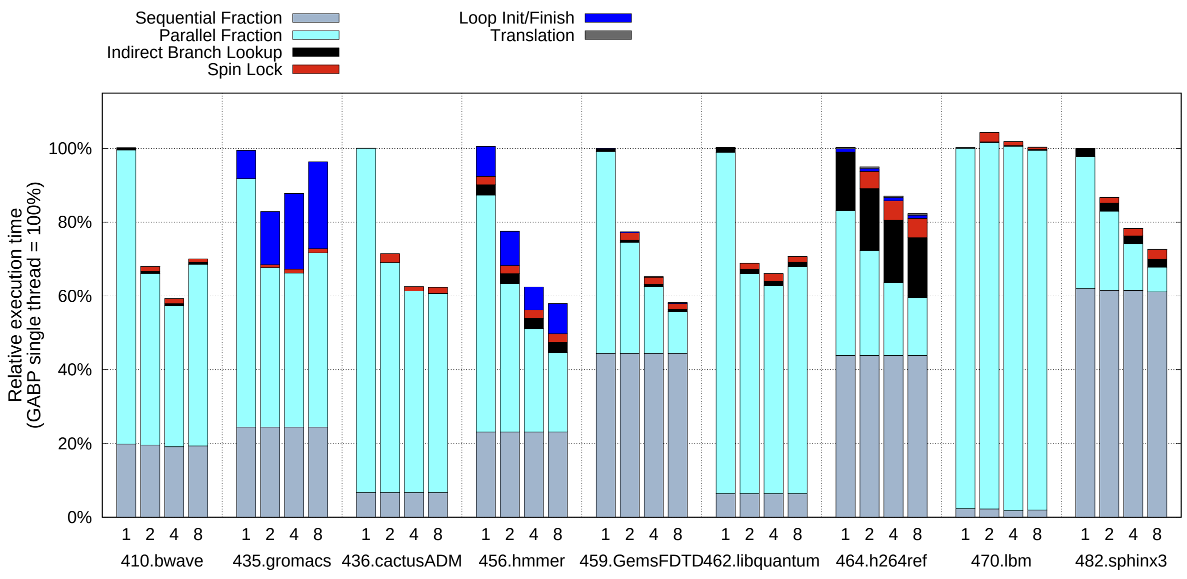


Figure 6.5: Breakdown of execution time for benchmark with `DOALL_BLOCK` loops on 1, 2, 4 and 8 threads on Intel(R) Core(TM) i7-3770K.

there is an apparent non-scaling parallel fraction for 2 and 4 threads. Even if they are fully independent pieces of code executed in parallel, the "uncore" part of the system prevents a linear scalability as the number of threads increases. The performance disparity between i7 and Xeon processors is partially due to the difference in interconnect, the last-level cache L3 and the external DRAM bandwidth.

For memory-bound applications such as `470.lbm` and `410.bwave`, the pressure on the capacity of the last level cache is proportional to the number of threads. We use the Linux `perf` tool to monitor cache statistics when running GABP and found that the number of last-level cache (LLC) references and misses for i7 CPU increased proportionally as the number of threads increased. This is different for the Xeon CPU. When increasing the number of threads in Xeon, we observed a linear increase in LLC references but roughly constant number of cache misses. This suggests that the cache capacity plays a critical role in terms of performance. The i7 processor has a 8MB L3 cache, making it difficult to cache the context of loop iterations for more than two threads at the same time compared to the Xeon processor whose 25MB L3 cache has a large enough capacity to exploit locality with eight threads. What's worse, due to the requirement of thread privatisation, the cache capacity is polluted with thread-private data, making the number of last-level cache references even higher. Moreover, given the hardware specification that the maximum memory bandwidth for the i7 is 25.6GB/s, just one third of the 76.8GB/s for the Xeon processor, the latency for memory stalling is much higher for the i7 processor. Last but not least, for the i7 processor, there are only 4 actual cores with 8 hardware threads available through hyperthreading. It seems the extra thread contexts do not bring the parallel performance improvement but rather have a negative performance impact on the system.

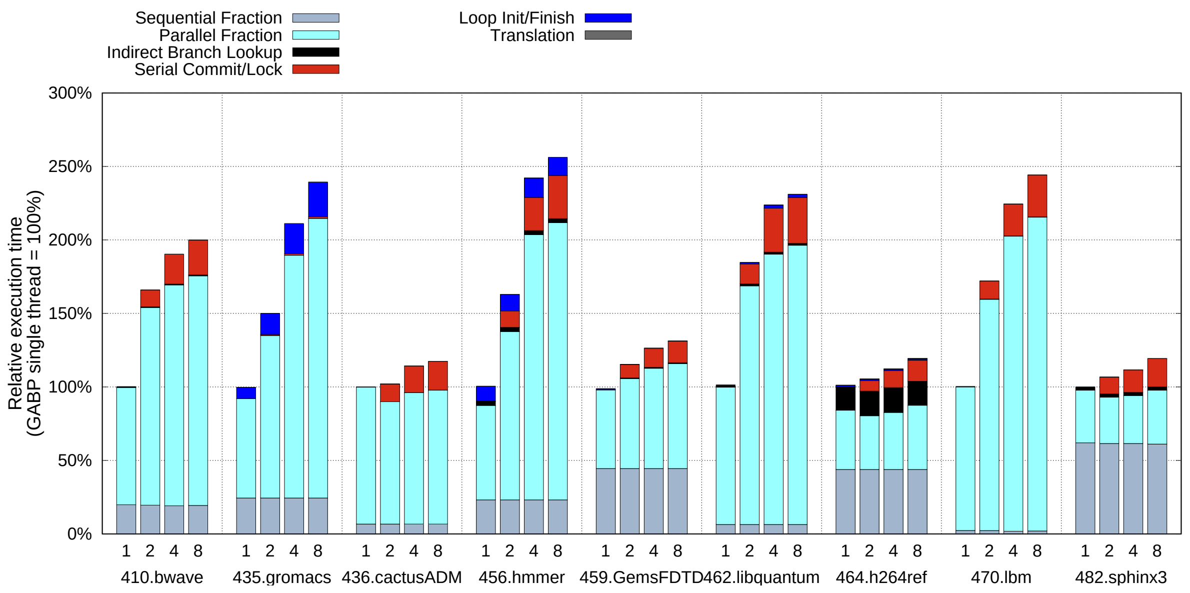


Figure 6.6: Breakdown of execution time for benchmark that are parallelised using the `DOALL_CYCLIC` approach on 1, 2, 4 and 8 threads on Intel(R) Xeon(R) E5-2667.

6.2.3 Cyclic vs Block Parallelisation

The second parallelisation approach for `DOALL` loops is the `DOALL_CYCLIC` approach that schedules threads to take turns to execute one iteration in a round robin order. The detailed parallelisation methods are discussed in Section 5.4.2. The `DOALL_CYCLIC` approach is also applicable for `DOALL` loops with unclear iteration counts at the entry of the loop. Figure 6.6 shows the execution breakdown of the `DOALL` loops to be parallelised in the cyclic approach.

GABP results in substantial slowdown if `DOALL` loops are parallelised cyclically. Compared to the execution time breakdown for `DOALL_BLOCK` parallelisation in Figure 6.3, two sources of overhead are significantly increased. The first overhead is the serialisation of threads during thread commit. Each thread is required to wait for the previous thread to commit in order to commit and work on the next iteration. The extra time is reflected in the increased time in the spin locks. As the number of threads increases, the time spent in spin lock increases .

Figure 6.3 also shows that the time spent in the parallel fraction increases, which violates the expectation of a reduction in time from parallelisation. As there is no other modification in the parallel fraction and the capacity of the cache is large enough for the Xeon processor, the performance degradation could be caused by a false sharing effect between threads. For threads that work on iterations in round robin order, it is likely that two adjacent threads would write to the same cache line. Granted there are no actual conflicts between the writes, but due to the requirement of the cache coherence protocol, threads would compete to invalidate other threads' L1 and L2 caches. Therefore it would result in frequent cache misses and causes a severe performance degradation. For cyclic based parallelisation, there exists a large fraction of accesses that would result in false sharing. For example a common integer array access with linear array indexes such as `a[i] = i`, an adjacent thread that writes to `a[i+1] = i+1` would inevitably result in the false cache sharing effect among threads.

The false sharing effect is the major barrier to prevent `DOALL_CYCLIC` parallelisation

False Sharing Test (FS)

```
//array initialised
float a[N], b[N];
for (i=0; i<N; i++) {
    b[i] = sqrt(a[i]);
}
```

No False Sharing Test (NoFS)

```
//array initialised
float a[8*N], b[8*N];
for (i=0; i<N; i++) {
    b[i*8] = sqrt(a[i*8]);
}
```

Figure 6.7: Test code for verifying the impact of false sharing

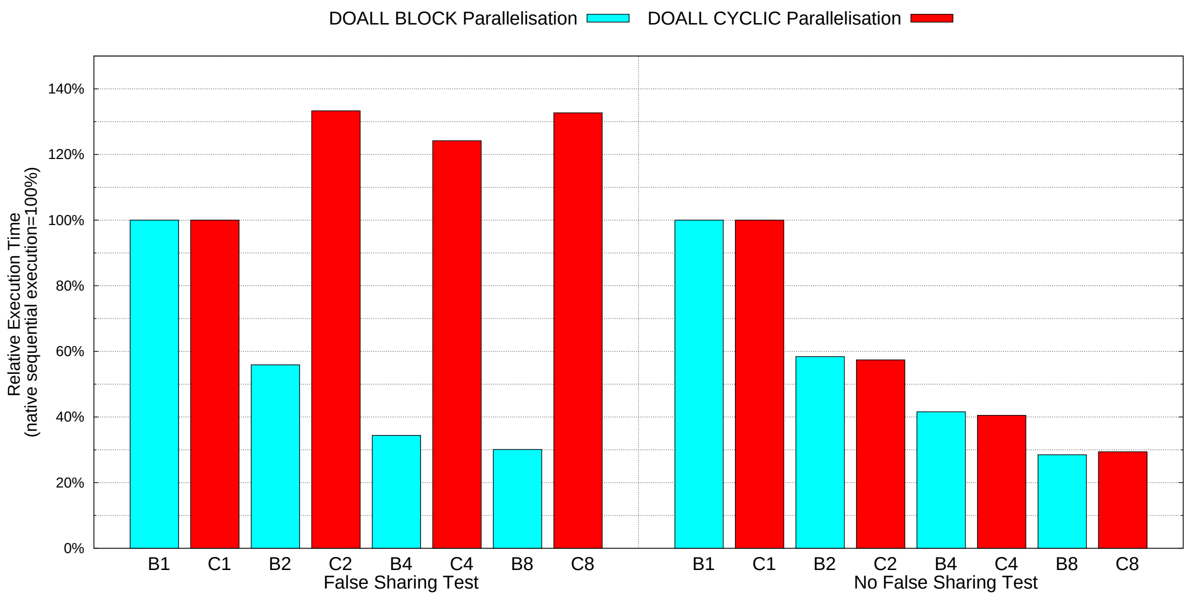


Figure 6.8: Measured relative execution time for the FS and NoFS tests using the DOALL_BLOCK and DOALL_CYCLIC parallelisation with 1, 2, 4 and 8 threads on Intel(R) Xeon(R) E5-2667.

from achieving desirable performances. To verify this hypothesis, I wrote two simple tests shown in Figure 6.7. The first test (FS) contains a simple loop that updates a single float variable at each iteration. As the `float` is 32-bit, adjacent iterations in this test may share the same cache line. While the second test (NoFS) has the same loop structure but updates an exclusive cache line at each iteration. Therefore the second test should not generate the false sharing effect during parallelisation. Both tests are compiled by `gcc` and −`O3` flags. Their executables are parallelised by GABP using both DOALL_CYCLIC and DOALL_BLOCK respectively. Figure 6.8 shows the performance difference between the two tests. For DOALL_CYCLIC parallelisation on the FS test, it results in a performance slowdown even with a larger number of threads. In contrast, the NoFS test has no cache line sharing between iterations and DOALL_CYCLIC achieves significant linear speedup. The performance is similar to the results from DOALL_BLOCK parallelisation. These two tests suggest that the false sharing effect is strongly related to the performance degradation in DOALL_CYCLIC parallelisation.

One effective solution to remove false sharing is to unroll the loop so that each thread exclusively modifies data structure elements greater than one cache line. In order to

Benchmark	ID	Contained Function	Coverage	Method	Estimated Loop Speedup (8 cores)	Average Transaction Size
401.bzip2	53	`mainSort`	35.6%	SPEC	5.12	525.6
429.mcf	23	`price_out_impl`	56.9%	SPEC	3.82	32.4
429.mcf	48	`primal_bea_mpp`	19.6%	SYNC	5.83	4
473.astar	113	`ZN6wayobj10makebound2EP`	21.3%	SPEC	2.42	18.2
482.sphinx	17	`approx_cont_mgau`	36.2%	SPEC	1.82	115.4

Table 6.3: Irregular loops selected for evaluating thread-level speculation and speculative synchronisation

determine the minimum factor of loop unrolling, the original data structure has to be recognised. The current implementation of static binary analysis lacks the features to retrieve data structure information. Due to the substantial implementation effort, the automatic hint instruction generation to guide loop unrolling has not yet been implemented. In the future, a study deciding the unroll factor in static binary analysis to guide loop unrolling is one of the future objectives to improve performance.

6.3 Irregular Loop Evaluation

In this section, I evaluate the ability to handle runtime cross-iteration dependencies described in Section 5.5. We aim to parallelise a wider range of loops with cross-iteration dependencies that are not easily removed using induction and reduction optimisations. To answer the question of whether irregular cross-iteration dependencies can be efficiently enforced using thread-level speculation and value prediction discussed in Section 5.6, a range of representative loops are selected for parallelisation shown in Table 6.3. The loops are selected based on their exhibition of irregular cross-iteration dependence characteristics from profiling information in BEEP. They also demonstrate beneficial parallelism from parallel execution cost models.

There exist other loops that demonstrate potential parallelism from the parallel execution model. However these loops exhibit many corner cases that are not well addressed by the current GABP implementation. Many loops contain pointer arithmetic, conditional execution or even nested indirect memory accesses. A tiny mis-speculation on these accesses may easily lead to a segmentation, arithmetic or bus fault, which complicates the process to recover the correct thread state for re-execution. Moreover, some mis-speculation may not lead to a fault, but it may jump to a non-transactional area and damage other contexts. The live fault is much more difficult to detect and prevent at runtime. The engineering effort to remove these corner cases is beyond the scope of the current research objective for this dissertation.

Figure 6.9 shows the estimated and measured speedup of the whole program while only parallelising the selected loops with 1, 2, 4 and 8 threads on the Xeon processor. The estimated speedups are derived from the corresponding realistic speculation and synchronisation models from BEEP. For the results, the actual parallel performance differs vastly compared to predicted performance. The measured execution shows a major slowdown of around 2× to 3×, even when the coverage of most parallelised loops is less than 40%. As the number of threads increases, the slowdown becomes worse. From the results,

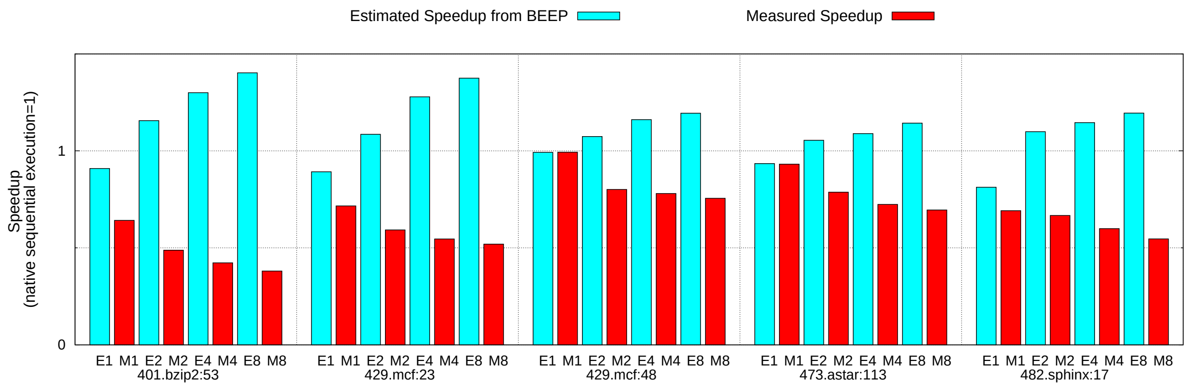


Figure 6.9: Estimated and measured whole-program speedups of benchmarks from the SPEC CPU2006 achieved by GABP, using two, four, or eight cores on the Intel(R) Xeon(R) E5-2667 processor. Only one loop is selected for parallelisation

it is found that the current GABP implementation for runtime dependence handling is insufficient to deliver performance improvement.

To investigate the factors that cause the observed slowdown in the actual execution on real systems, I use the same sampling approach for overhead analysis as discussed in Section 6.2.1. Compared to measuring the overhead from `DOALL` parallelisation, loops parallelised using thread-level speculation suffer a large variation in execution time as well as overhead breakdown due to the nondeterministic nature of speculative execution. Based on the specification of thread-level speculation, the sampled PC logs are divided into seven clusters.

- Sequential fraction: the fraction of time for executing the serial part of the executable. It is the part that is not optimised.
- Parallel fraction: the fraction of time for parallel execution of the selected loops in the original application. However it is not easy to differentiate the interrupted PC as being from first-time execution or re-execution after a mis-speculation. Therefore the parallel fraction also includes the overhead for re-execution of the transaction.
- Check Points: the fraction of time for threads to save/restore machine contexts to/from its check-point buffer during start/rollback of a transaction.
- Hash Table Lookup: the time for each thread to redirect its first read to its read set and buffer all its writes to its write set.
- STM Validate/Commit/Clear: the time for each thread to validate all entries from its read set against shared memory and the time to commit the buffered writes to shared memory if validation is successful. After commits, the threads clears the content of its read and write buffers. It also includes the time for clearing the transaction if the validation fails.
- Spin lock overhead: the waiting time in spin locks for threads to commit in order and the time for forwarding values between threads are also included.

Figure 6.10 shows the calculated time breakdown for the five loops. From the graph it can be seen that the time spent in `JIT_STM` is roughly the same as the time spent

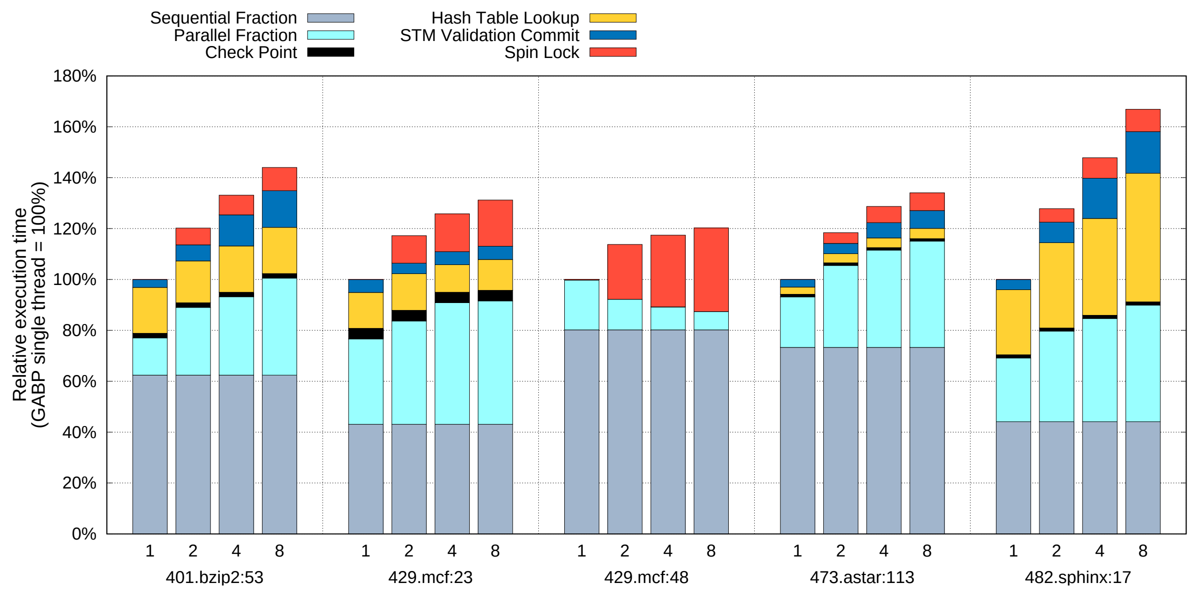


Figure 6.10: Breakdown of execution time for benchmark that are parallelised using the DOACROSS_SPECULATION approach on 1, 2, 4 and 8 threads on Intel(R) Xeon(R) E5-2667.

in the parallel fraction of the original code. The top source of JIT_STM overhead is the hash table lookups where speculative accesses are redirected to threads' read/write buffers. `429.mcf:48` is the only loop that uses hash table elision by JIT compiling direct memory accesses to fixed thread private write buffers. For other loops, there exist a large fraction of memory accesses that are not decided, therefore they are not applicable for hash table elision. The time spent in JIT_STM read validation and write commit is another large source of overhead. The overhead is also dependent on the actual size of the transaction for validation and commit. For large loops, the overhead for check-pointing is relatively small compared to other sources of overhead.

However, the time for STM validation/commit and the parallel fraction increases as the number of threads increases. It violates the expectation that the execution time should decrease as each thread cyclically executes only a fraction of total iteration space. From a real system, it is difficult to determine the precise reason why the time increases. We observed that the code for STM validation/commit is JIT generated as straight lines of memory reads and writes in a critical section. A possible reason for slowdown in STM validation/commit is due to cache misses during read validation and potential false sharing effects during the write commits. Similarly for the time spent in parallel fraction grows as the number of threads increases, which bucks the expectation of time reduction benefits from parallelisation. The increase in time might be caused by cache misses from false sharing between non-speculative shared writes. The false sharing penalty is also amplified by re-execution of the loop code in case of mis-speculation.

The only exception in the parallel fraction is the `429.mcf:48` test that uses synchronisation instead of speculation. We can see the parallel fraction decreases as the number of threads grows. This is due to the fact that the loop in `429.mcf:48` only performs writes for 6% of the total time. The false sharing effect is less significant. However with greater numbers of threads, the time spent in locks increases rapidly, negating the benefits brought by parallelisation.

The take away from speculation overhead analysis is that false sharing from a coherent

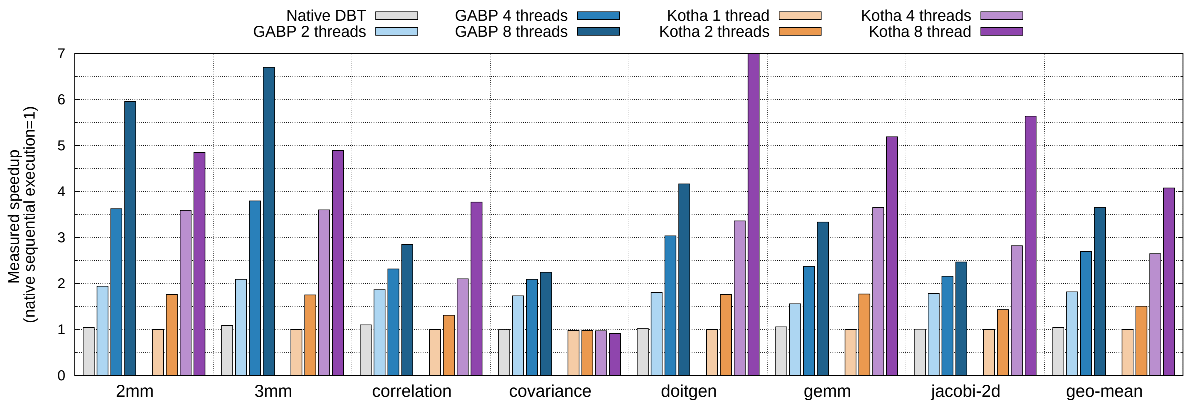


Figure 6.11: Whole-program speedups of benchmarks from the PolyBench achieved by GABP, using two, four, or eight cores on the Intel(R) Xeon(R) E5-2667 processor. The reference performance is from the Kotha's binary parallelisation work [132]

cache hierarchy impacts significantly on the ability to use thread-level speculation for parallelisation. The effect of cache misses could not be simply modelled in the BEEP parallel execution models nor other limit studies from other work, making the estimation of loop performance inaccurate. This is because the modelling of the cache hierarchy under a parallel environment is highly non-deterministic and varies wildly for different systems. BEEP is only meant to be general and not specific to a single implementation. To address the problem of cache misses, more efficient and cache friendly implementations of `JIT_STM` are required. Loops have to be unrolled so that transactions can take the whole cache line as much as possible.

Without explicit hardware extension to support thread-level speculation, it is very difficult to achieve performance through software-based implementation on existing structure of cache hierarchies. Even with optimised solution from JIT compilation, the overhead of speculative execution is still around 2x slowdown compared to native execution.

6.4 Performance Comparison and Related Work

In this section, performance is compared with other existing parallelisation frameworks to demonstrate the strength of dynamic binary parallelisation using GABP. We only compare the performance with the work that demonstrates performance on real systems. We do not compare our approach with simulated results such as RASP [134] since they assume hardware support that has not been implemented in any current processor.

6.4.1 Comparison with Kotha on PolyBench

Kotha et al. [132] proposed a static binary parallelisation tool that only parallelises affine loops and rewrites the input binary in their SecondWrite binary rewriter. They achieved substantial performance on the Polybench [136] benchmark. To compare the GABP performance with their work, the parallelisation is also evaluated on the Polybench benchmark. However, their binary paralleliser is not publicly available. We contacted the authors who confirmed they had no plans to release it. Therefore performance for their static parallelisation tool is based on the reported results from their paper [132] . Figure

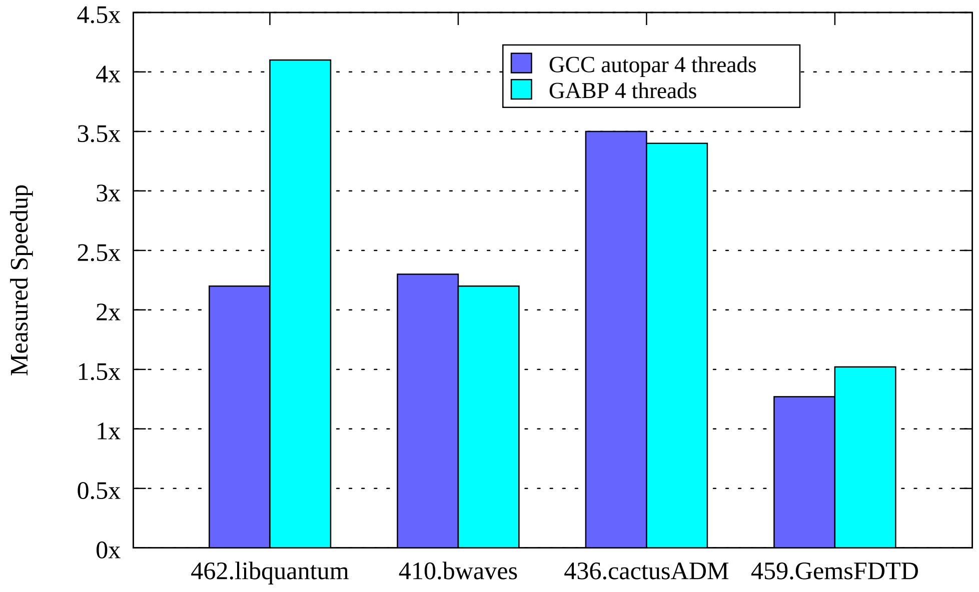


Figure 6.12: Whole-program speedups of benchmarks achieved by `gcc5.4` and GABP respectively, using four cores on the Intel(R) Xeon(R) E5-2667 processor.

6.11 shows the performance comparison between the speedup achieved by GABP and Kotha's tool. GABP achieves a geometric mean speedup of 2.7x for 4 threads and 3.6x for 8 threads, while Kotha has the geometric mean of 2.6x for 4 threads and 4.1x for 8 threads.

Although it is not a fair comparison as the performance for Kotha is obtained from a different machine, it illustrates that GABP achieves similar performance compared to Kotha's work. In Kotha's work [132], they claimed that dynamic binary parallelisation is sub-optimal due to translation overhead. The results from GABP show that the dynamic overhead is negligible for most programs especially benchmarks with high parallelism due to runtime optimisations. Sometimes dynamic binary parallelisation may outperform static parallelisation for some affine programs such as matrix multiplication `2mm` and `3mm`.

We also observe that the compiled executables from the Polybench are very simple. Each benchmark executable only contains one or a small number of loops with fully decided dependencies and loop iteration count, which is required for affine loops. However, for general application binaries, affine loops are hardly seen despite many efforts to extend its applicability [63]. Compared with Kotha's tool, GABP is able to parallelise more types of loop from complicated benchmarks with much broader applicability.

6.4.2 Comparison with `gcc autopar`

To compare automatic parallelisation at binary and compiler levels, performance is also compared with executables generated from conventional compilers from the same source code. The open source compiler `gcc` is the most easily accessed compiler that integrates an automatic parallelisation pass [137] for a limited range of `DOALL` loops.

We use `gcc5.4` with the `autopar` flag `-floop-parallelize-all` and generate executables for 4 threads `-ftree-parallelize-loops=4`. Figure 6.12 illustrates the performance comparison between `gcc autopar` and GABP for 4 threads. It shows that GABP binary parallelisation achieves similar performance to the `gcc` automatic parallelisation passes at source code level. It proves that access to source code is not a strict requirement to enable automatic parallelisation and achieve performance.

The performance difference of `462.libquantum` is due to the selection of different loops for parallelisation. GABP selects the loops based on its parallel execution model from BEEP profiling, while `gcc autopar` integrates its own cost model for parallelisation. Moreover, thanks to the trace optimisation brought by dynamic binary translation, the final performance is beyond the theoretical limit of $4\times$.

6.5 Summary

In this chapter, the actual performance of the proposed automatic binary paralleliser GABP is evaluated. A geometric mean of $1.91\times$ speedup is achieved on a selection of benchmarks on real machines with eight threads. Firstly, the performance from GABP demonstrates that automatic parallelisation at a binary level is effective. GABP is also source and compiler agnostic. It can parallelise binaries from different languages and compilers.

Secondly, from overhead analysis, it is found that variations in machine specifications may have a substantial impact on parallel performance. However the performance differences are not directly derived from the proposed GABP code modification in this dissertation but the nature of executing parallel applications on a multi-core system. Cyclic `DOALL` parallelisation suffers significantly from false sharing effects. With further engineering work and fine-tuning the performance optimisation, these problems could be resolved.

Thirdly, the current implementation of GABP for thread-level speculation and synchronisation results in vast slowdown. The false sharing effect may also be the prime reason for the slowdown.

Lastly, by comparing with other related work, it demonstrates that GABP achieves similar or better performance. Compared to static binary parallelisation, the overhead of dynamic binary translation is negligible. And thanks to dynamic binary parallelisation, the scope of loops from GABP is much wider. By comparing with automatic parallelisation from gcc, it proves that access to source code is not a strict requirement to enable automatic parallelisation.

Chapter 7

Conclusion and Future Work

Every year new generations of hardware have constantly been released into the market. Now, commodity desktop and server class processors with 8 to 16 cores are the norm. As much existing software has been labelled as “legacy” due to replacement of new hardware, it is important not to overlook the fact that sequential performance of the legacy software becomes increasingly sub-optimal. In this dissertation, I argue that it is more cost-effective to directly optimise the original application binaries on new generations of hardware, especially through techniques like automatic parallelisation. The GABP tool, proves the hypothesis and demonstrates actual performance gain on existing hardware systems.

7.1 Contribution

In this dissertation, three major and novel contributions are presented to address the challenge of optimising legacy binaries.

7.1.1 Guided Binary Recompilation

The first contribution is the static-dynamic approach to enable complex and efficient binary recompilation. Our GBR (Guided Binary Recompilation) tool is implemented to recompile and transform stripped application binaries without the need for the source code. GBR performs static binary analysis to determine how recompilation should be undertaken, and produces a domain-specific hint program. The hint program is loaded and interpreted by GBR to guide a dynamic binary translator for recompilation. The novelty of the approach is to shift the majority of the complexity of dynamic binary recompilation into a static compilation problem and hint program generation. Dynamic binary recompilation can be simplified to virtual machines that interpret hint instructions.

GBR combines the strengths of both static analysis and dynamic JIT optimisation. With the expressive power of hint programs, it provides an open platform to automatically apply sophisticated optimisation transformations that were previously not applicable for legacy binaries. In chapter 3, I use two case studies of automatic software pre-fetching and vectorisation to demonstrate the effectiveness of GBR. They achieve significant performance improvement from prefetching and vectorisation on real systems.

By only recompiling the hot region of a highly optimised binary, the performance from GBR could even exceed the performance from the pre-compiled binary with the same optimisation transformations at an IR level. Our experiments from chapter 3 showed that the order and output of an optimisation pass in a compiler may limit or affect other optimisation passes, while there is no such problem when directly optimising binaries in GBR.

7.1.2 Binary Emulator for Estimating Parallelism

The second contribution is the proposal of the BEEP tool (Binary Emulator for Estimating Parallelism), an extension to GBR for guided binary instrumentation. BEEP is used to identify potential thread-level parallelism through static binary analysis and binary instrumentation. Two novel aspects of BEEP are:

- `Demand-driven Analysis:` BEEP performs preliminary static analysis on binaries and encodes all statically-undecided questions into hint programs. The questions are then answered at runtime by collecting the information through instrumentation with training inputs. The answers are then sent back for more accurate static analysis.
- `Parallel execution models:` BEEP incorporates parallel execution models to evaluate identified parallelism under different parallelisation paradigms and runtime conditions. The models are calculated based on the events generated from both the hint program and runtime content.

Compared to other binary instrumentation frameworks, BEEP enables high-level and on-demand instrumentation thanks to the prior static analysis information from hint programs. Compared to other compiler IR-based instrumentation, analysis from BEEP achieves the same expressive power as in IR and it can also accurately capture runtime events as it instruments the final released original binary.

In Chapter 4, I discuss three ideal models and two realistic parallel cost models to uncover the thread-level parallelism from binaries. From the three ideal parallel models of data-flow, code motion and induction reduction optimisations, the majority of loops with low parallelism are filtered out. Then I evaluate the remaining loops with models of thread-level speculation and synchronisation with estimated overhead costs and hardware constraints. It is found that the removal of cross-iteration dependencies from induction/reduction analysis and value prediction brings much more benefit than simply enforcing dependencies using speculation and synchronisation.

7.1.3 Guided Automatic Binary Parallelisation

The third significant contribution is GABP (Guided Automatic Binary Parallelisation), an extension to GBR for automatic binary parallelisation. GABP focuses on loops from sequential application binaries and automatically extracts thread-level parallelism from them on-the-fly, under the direction of the hint program, for efficient parallel execution. It employs a hybrid of parallelisation schemes based on the recognised type of loops. For `DOALL` loops, it performs JIT optimisations for induction and reduction variables for different threads. Privatisation is handled with minimum overhead through JIT compilation on thread-private code caches. For loops with cross-iteration dependencies, GABP employs `JITSTM` to support thread-level speculation and speculative synchronisation to maintain correct execution. GABP achieves a geometric mean of speedup of 1.91$\times$ on binaries from SPEC CPU2006 on a real x86-64 eight-core system compared to native sequential execution.

7.2 Future Work

In this dissertation, the framework of GBR is laid out as the fundamental infrastructure for optimising legacy binaries. The implementation of GBR is still in its prototype stage and far from complete. There are a few interesting directions for future work and research.

7.2.1 Standardisation

In the future, I aim to provide more optimisation and analysis tools on top of GBR and standardise the interface of hint program specification (hint ISA) for compatibility. The following lists of extensions that are proposed for GBR; each extension constitutes a substantial research project:

- **Automatic Binary Prefetcher**: the tool performs static analysis on the input binary, recognises potential opportunities for software pre-fetching. The tool leaves static hints for guiding the dynamic binary translator to insert hardware prefetching instructions at specified locations.
- **Automatic Binary Vectoriser**: the tool performs static analysis on the input binary, recognises potential loops for vectorisation. The tool leaves static hints for guiding the dynamic binary translator to recompile the specified loop into a vectorised format using conventional algorithms and the latest hardware SIMD extensions.
- **Automatic Function Inliner**: the tool performs static analysis on the input binary, recognises potential functions for inlining. The tool leaves static hints to improve the trace creation process in dynamic binary translation.
- **Automatic Lock Elision**: the tool performs static analysis on the input binary, recognises potential code for locks and critical sections. The tool leaves static hints to rewrite the lock into `JITSTM` speculative execution.

7.2.2 Static Binary Analysis

As discussed throughout the dissertation, the largest limiting factor of GBR is the accuracy of static binary analysis, it is essential to develop a powerful static binary analysis tool that conforms to the protocol of the hint ISA and understands the nature of dynamic binary translation. The following lists the required features to facilitate robust binary recompilation:

- **Binary alias analysis**: the current tool is not able to tell whether two memory accesses alias or not, it therefore generates unnecessary speculative code to maintain correct execution.
- **Loop variable analysis**: the tool needs to decide the most efficient way to handle the loop variables in case of induction, reduction or private variables.
- **Removing cross-iteration dependencies**: the tool needs to understand the data structure and decide an optimal solution to remove cross-iteration dependencies, through value prediction, array privatisation and algorithm-level optimisation.

- **Decompiling to compiler IR**: the tool can decompile part of a binary into a compiler IR, such as LLVM IR, for more powerful analysis and code generation. The static tool should maintain consistency between the decompiled code and the rest of binary code.

7.2.3 Adaptive Runtime System

The runtime system consists of the GBR virtual machine (GVM) that interprets hint instructions, transforms each basic block and buffers the modified code in thread-private code caches. In the future, a more robust adaptive binary recompilation can be implemented for more aggressive runtime optimisation. Adaptation can be realised by generating different copies of the same basic block each with different modifications. A dynamic switch can be placed during linking. By checking runtime conditions, the most efficient modification can be selected for later execution.

Bibliography

[1] Herb Sutter. The free lunch is over: A fundamental turn toward concurrency in software.

[2] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. In *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA '11, pages 365–376, New York, NY, USA, 2011. ACM.

[3] Bradford Nichols, Dick Buttlar, and Jacqueline Farrell. *Pthreads programming: A POSIX standard for better multiprocessing.* " O'Reilly Media, Inc.", 1996.

[4] Message P Forum. MPI: A message-passing interface standard. Technical report, Knoxville, TN, USA, 1994.

[5] Mitsuhisa Sato. OpenMP: parallel programming API for shared memory multiprocessors and on-chip multiprocessors. In *System Synthesis, 2002. 15th International Symposium on*, pages 109–111. IEEE, 2002.

[6] James Reinders. *Intel threading building blocks: outfitting C++ for multi-core processor parallelism.* " O'Reilly Media, Inc.", 2007.

[7] Charlene O'Hanlon. A conversation with David Brown. *Queue*, 4(8):14–23, 2006.

[8] Lorin Hochstein, Jeffrey Carver, Forrest Shull, Sima Asgari, Victor Basili, Jeffrey K Hollingsworth, and Marvin V Zelkowitz. Parallel programmer productivity: A case study of novice parallel programmers. In *Supercomputing, 2005. Proceedings of the ACM/IEEE SC 2005 conference*, pages 35–35. IEEE, 2005.

[9] Simone Campanoni, Timothy Jones, Glenn Holloway, Vijay Janapa Reddi, Gu-Yeon Wei, and David Brooks. HELIX: automatic parallelization of irregular programs for chip multiprocessing. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, pages 84–93. ACM, 2012.

[10] Guilherme Ottoni, Ram Rangan, Adam Stoler, and David I. August. Automatic thread extraction with decoupled software pipelining. In *Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 38, pages 105–118, Washington, DC, USA, 2005. IEEE Computer Society.

[11] Robert P Wilson, Robert S French, Christopher S Wilson, Saman P Amarasinghe, Jennifer M Anderson, Steve WK Tjiang, Shih-Wei Liao, Chau-Wen Tseng, Mary W Hall, Monica S Lam, et al. SUIF: An infrastructure for research on parallelizing and optimizing compilers. *ACM Sigplan Notices*, 29(12):31–37, 1994.

[12] Uday Bondhugula, J. Ramanujam, and P. Sadayappan. PLuTo: A practical and fully automatic polyhedral parallelizer and locality optimizer. Technical Report OSU-CISRC-10/07-TR70, The Ohio State University, October 2007.

[13] Tobias Grosser, Hongbin Zheng, Raghesh Aloor, Andreas Simbürger, Armin Größlinger, and Louis-Noël Pouchet. Polly-Polyhedral optimization in LLVM.

[14] Cristina Cifuentes. Binary translation: Static, dynamic, retargetable? In *Software Maintenance 1996, Proceedings., International Conference on*, 1996.

[15] Sorav Bansal and Alex Aiken. Binary translation using peephole superoptimizers. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, pages 177–192. USENIX Association, 2008.

[16] Ted Romer, Geoff Voelker, Dennis Lee, Alec Wolman, Wayne Wong, Hank Levy, Brian Bershad, and Brad Chen. Instrumentation and Optimization of Win32/Intel Executables Using Etch. In *Proceedings of the USENIX Windows NT Workshop on The USENIX Windows NT Workshop 1997*, NT'97, pages 1–1, Berkeley, CA, USA, 1997. USENIX Association.

[17] L. Van Put, D. Chanet, B. De Bus, B. De Sutter, and K. De Bosschere. DIABLO: a reliable, retargetable and extensible link-time rewriting framework. In *Signal Processing and Information Technology, 2005. Proceedings of the Fifth IEEE International Symposium on*, pages 7–12, Dec 2005.

[18] Alan Eustace and Amitabh Srivastava. ATOM: A flexible interface for building high performance program analysis tools. In *Proceedings of the USENIX 1995 Technical Conference Proceedings*, TCON'95, pages 25–25, Berkeley, CA, USA, 1995. USENIX Association.

[19] Brian Walters. VMware virtual platform. *Linux journal*, 1999(63es):6, 1999.

[20] Eric Traut. Building the virtual pc. *Byte*, 22(11):51–52, 1997.

[21] Derek Bruening. Efficient, transparent, and comprehensive runtime code manipulation. Technical report, Massachusetts Institute of Technology, 2004. Ph.D. Thesis.

[22] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 190–200, New York, NY, USA, 2005. ACM.

[23] Amanieu D'Antras, Cosmin Gorgovan, Jim Garside, and Mikel Luján. Low overhead dynamic binary translation on ARM. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 333–346. ACM, 2017.

[24] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *ACM Sigplan notices*, volume 42, pages 89–100. ACM, 2007.

[25] Fabrice Bellard. QEMU, a fast and portable dynamic translator.

[26] Xiaochun Zhang, Qi Guo, Yunji Chen, Tianshi Chen, and Weiwu Hu. HERMES: a fast cross-ISA binary translator with post-optimization. In *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 246–256. IEEE Computer Society, 2015.

[27] Emilio G Cota, Paolo Bonzini, Alex Bennée, and Luca P Carloni. Cross-ISA machine emulation for multicores. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization*, pages 210–220. IEEE Press, 2017.

[28] Kemal Ebcioğlu and Erik R Altman. DAISY: Dynamic compilation for 100% architectural compatibility. In *ACM SIGARCH Computer Architecture News*, volume 25, pages 26–37. ACM, 1997.

[29] Derek Lane Bruening. *Efficient, transparent, and comprehensive runtime code manipulation.* PhD thesis, Massachusetts Institute of Technology, 2004.

[30] Michael D Bond and Kathryn S McKinley. Practical path profiling for dynamic optimizers. In *Proceedings of the international symposium on Code generation and optimization*, pages 205–216. IEEE Computer Society, 2005.

[31] Emmett Witchel and Mendel Rosenblum. Embra: Fast and flexible machine simulation. In *ACM SIGMETRICS Performance Evaluation Review*, volume 24, pages 68–79. ACM, 1996.

[32] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: a transparent dynamic optimization system. In *ACM SIGPLAN Notices*, volume 35, pages 1–12. ACM, 2000.

[33] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. An infrastructure for adaptive dynamic optimization. In *Code Generation and Optimization, 2003. CGO 2003. International Symposium on*, pages 265–275. IEEE, 2003.

[34] Derek Bruening. DR coverage. `http://dynamorio.org/docs/page_drcov.html`.

[35] Derek Bruening and Qin Zhao. Practical memory checking with Dr. Memory. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 213–223. IEEE Computer Society, 2011.

[36] Julian Seward and Nicholas Nethercote. Using Valgrind to detect undefined value errors with bit-precision. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, pages 2–2, Berkeley, CA, USA, 2005. USENIX Association.

[37] Vladimir Kiriansky, Derek Bruening, Saman P Amarasinghe, et al. Secure execution via program shepherding.

[38] Intel. Intel Parallel Inspector. `http://software.intel.com/en-us/intel-parallel-inspector/`.

[39] Jeanne Ferrante, Karl J Ottenstein, and Joe D Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 9(3):319–349, 1987.

[40] Saumya Debray, Robert Muth, and Matthew Weippert. Alias analysis of executable code. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 12–24. ACM, 1998.

[41] Gogul Balakrishnan and Thomas Reps. Analyzing memory accesses in x86 executables. In *International conference on compiler construction*, pages 5–23. Springer, 2004.

[42] Susan Horwitz, Jan Prins, and Thomas Reps. On the adequacy of program dependence graphs for representing programs. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 146–157. ACM, 1988.

[43] Ron Cytron, Jeanne Ferrante, Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(4):451–490, 1991.

[44] Ken Kennedy. *A survey of data flow analysis techniques.*

[45] William Landi. Undecidability of static analysis. *ACM Letters on Programming Languages and Systems (LOPLAS)*, 1(4):323–337, 1992.

[46] Lars Ole Andersen. *Program analysis and specialization for the C programming language.* PhD thesis, 1994.

[47] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 32–41. ACM, 1996.

[48] John Whaley and Monica S Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *ACM SIGPLAN Notices*, volume 39, pages 131–144. ACM, 2004.

[49] Bolei Guo, Matthew J Bridges, Spyridon Triantafyllis, Guilherme Ottoni, Easwaran Raman, and David I August. Practical and accurate low-level pointer analysis. In *Proceedings of the international symposium on Code generation and optimization*, pages 291–302. IEEE Computer Society, 2005.

[50] Nick P. Johnson, Jordan Fix, Stephen R. Beard, Taewook Oh, Thomas B. Jablin, and David I. August. A collaborative dependence analysis framework. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization*, CGO 2017, pages 148–159, Piscataway, NJ, USA, 2017. IEEE Press.

[51] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Solving shape-analysis problems in languages with destructive updating. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 20(1):1–50, 1998.

[52] Susan Horwitz. Precise flow-insensitive may-alias analysis is NP-hard. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 19(1):1–6, 1997.

[53] Ken Kennedy and John R. Allen. *Optimizing compilers for modern architectures: a dependence-based approach.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

[54] Ron Cytron. Doacross: Beyond vectorization for multiprocessors. In *ICPP*, pages 836–844, 1986.

[55] James Russell Beckman Davies. Parallel loop constructs for multiprocessors. Technical report, University of Illinois at Urbana-Champaign, 1981. M.S. Thesis.

[56] Bill Blume, Rudolf Eigenmann, Keith Faigin, John Grout, Jay Hoeflinger, David Padua, Paul Petersen, Bill Pottenger, Lawrence Rauchwerger, Peng Tu, and Stephen Weatherford. Polaris: The next generation in parallelizing compilers. In *Proceedings of the workshop on languages and compilers for parallel computing*, pages 10–1. Springer-Verlag, Berlin/Heidelberg, 1994.

[57] Lawrence Rauchwerger and David A Padua. The LRPD test: Speculative run-time parallelization of loops with privatization and reduction parallelization. *IEEE Transactions on Parallel and Distributed Systems*, 10(2):160–180, 1999.

[58] Peng Tu and David Padua. Automatic array privatization. In *Compiler optimizations for scalable parallel systems*, pages 247–281. Springer, 2001.

[59] Simone Campanoni, Timothy Jones, Glenn Holloway, Vijay Janapa Reddi, Gu-Yeon Wei, and David Brooks. HELIX: Automatic parallelization of irregular programs for chip multiprocessing. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, CGO '12, pages 84–93, New York, NY, USA, 2012. ACM.

[60] John L. Henning. SPEC CPU2006 Benchmark Descriptions. *SIGARCH Comput. Archit. News*, 34(4):1–17, September 2006.

[61] Easwaran Raman, Guilherme Ottoni, Arun Raman, Matthew J. Bridges, and David I. August. Parallel-stage decoupled software pipelining. In *Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '08, pages 114–123, New York, NY, USA, 2008. ACM.

[62] Jialu Huang, Arun Raman, Thomas B. Jablin, Yun Zhang, Tzu-Han Hung, and David I. August. Decoupled software pipelining creates parallelization opportunities. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '10, pages 121–130, New York, NY, USA, 2010. ACM.

[63] Mohamed-Walid Benabderrahmane, Louis-Noël Pouchet, Albert Cohen, and Cédric Bastoul. The polyhedral model is more widely applicable than you think. In *International Conference on Compiler Construction*, pages 283–303. Springer, 2010.

[64] Niall Murphy, Timothy Jones, Robert Mullins, and Simone Campanoni. Performance implications of transient loop-carried data dependences in automatically parallelized loops. In *Proceedings of the 25th International Conference on Compiler Construction*, CC 2016, pages 23–33, New York, NY, USA, 2016. ACM.

[65] Gurindar S Sohi, Scott E Breach, and TN Vijaykumar. Multiscalar processors. In *ACM SIGARCH Computer Architecture News*, volume 23, pages 414–425. ACM, 1995.

[66] Lance Hammond, Benedict A Hubbert, Michael Siu, Manohar K Prabhu, Michael Chen, and K Olukolun. The Stanford Hydra CMP. *IEEE Micro*, 20(2):71–84, 2000.

[67] Lance Hammond, Mark Willey, and Kunle Olukotun. Data speculation support for a chip multiprocessor. *SIGOPS Oper. Syst. Rev.*, 32(5):58–69, October 1998.

[68] J. Gregory Steffan, Christopher Colohan, Antonia Zhai, and Todd C. Mowry. The STAMPede approach to thread-level speculation. *ACM Trans. Comput. Syst.*, 23(3):253–300, August 2005.

[69] Wei Liu, James Tuck, Luis Ceze, Wonsun Ahn, Karin Strauss, Jose Renau, and Josep Torrellas. POSH: A TLS Compiler That Exploits Program Structure. In *Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '06, pages 158–167, New York, NY, USA, 2006. ACM.

[70] Carlos García Quiñones, Carlos Madriles, Jesús Sánchez, Pedro Marcuello, Antonio González, and Dean M. Tullsen. Mitosis compiler: An infrastructure for speculative threading based on pre-computation slices. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 269–279, New York, NY, USA, 2005. ACM.

[71] Tim Harris, James Larus, and Ravi Rajwar. *Transactional Memory, 2nd Edition.* Morgan and Claypool Publishers, 2nd edition, 2010.

[72] Ravi Rajwar and James R Goodman. Speculative lock elision: Enabling highly concurrent multithreaded execution. In *Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture*, pages 294–305. IEEE Computer Society, 2001.

[73] Amitabha Roy. *Software lock elision for x86 machine code.* PhD thesis, University of Cambridge, 2011.

[74] Maurice Herlihy and J Eliot B Moss. *Transactional memory: Architectural support for lock-free data structures*, volume 21. ACM, 1993.

[75] Dave Dice, Yossi Lev, Mark Moir, Dan Nussbaum, and Marek Olszewski. Early experience with a commercial hardware transactional memory implementation. 2009.

[76] Janice M Stone, Harold S Stone, Philip Heidelberger, and John Turek. Multiple reservations and the oklahoma update. *IEEE Parallel & Distributed Technology: Systems & Applications*, 1(4):58–71, 1993.

[77] Richard M. Yoo, Christopher J. Hughes, Konrad Lai, and Ravi Rajwar. Performance evaluation of Intel's transactional synchronization extensions for high-performance computing. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 19:1–19:11, New York, NY, USA, 2013. ACM.

[78] James Reinders. Transactional synchronization in Haswell. `http://software.intel.com/en-us/blogs/2012/02/07/transactional-synchronization-in-haswell/`, july 2012.

[79] Christian Jacobi, Timothy Slegel, and Dan Greiner. Transactional memory architecture and implementation for IBM System Z. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '12, pages 25–36, Washington, DC, USA, 2012. IEEE Computer Society.

[80] Sean Lie. *Hardware support for unbounded transactional memory.* PhD thesis, Citeseer, 2004.

[81] Peter Damron, Alexandra Fedorova, Yossi Lev, Victor Luchangco, Mark Moir, and Daniel Nussbaum. Hybrid transactional memory. In *ACM Sigplan Notices*, volume 41, pages 336–346. ACM, 2006.

[82] Dave Dice, Ori Shalev, and Nir Shavit. Transactional Locking II. In *International Symposium on Distributed Computing*, pages 194–208. Springer, 2006.

[83] Bratin Saha, Ali-Reza Adl-Tabatabai, Richard L Hudson, Chi Cao Minh, and Benjamin Hertzberg. McRT-STM: a high performance software transactional memory system for a multi-core runtime. In *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 187–197. ACM, 2006.

[84] Tim Harris, Mark Plesko, Avraham Shinnar, and David Tarditi. Optimizing memory transactions. *ACM SIGPLAN Notices*, 41(6):14–25, 2006.

[85] Marek Olszewski, Jeremy Cutler, and J Gregory Steffan. JudoSTM: A dynamic binary-rewriting approach to software transactional memory. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, pages 365–375. IEEE Computer Society, 2007.

[86] Zhao-Hui Du, Chu-Cheow Lim, Xiao-Feng Li, Chen Yang, Qingyu Zhao, and Tin-Fook Ngai. A cost-driven compilation framework for speculative parallelization of sequential programs. *ACM SIGPLAN Notices*, 39(6):71–81, 2004.

[87] Troy A Johnson, Rudolf Eigenmann, and TN Vijaykumar. Speculative thread decomposition through empirical optimization. In *Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 205–214. ACM, 2007.

[88] Hongtao Zhong, Mojtaba Mehrara, Steve Lieberman, and Scott Mahlke. Uncovering hidden loop level parallelism in sequential applications. In *High Performance Computer Architecture, 2008. HPCA 2008. IEEE 14th International Symposium on*, pages 290–301. IEEE, 2008.

[89] Peng Wu, Arun Kejariwal, and Călin Caşcaval. Compiler-driven dependence profiling to guide program parallelization. In *International Workshop on Languages and Compilers for Parallel Computing*, pages 232–248. Springer, 2008.

[90] Christoph Von Praun, Luis Ceze, and Calin Caşcaval. Implicit parallelism with ordered transactions. In *Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 79–89. ACM, 2007.

[91] Chadd C Williams and Jeffrey K Hollingsworth. Interactive binary instrumentation.

[92] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J Schwartz. BAP: A binary analysis platform. In *Computer Aided Verification*, pages 463–469. Springer, 2011.

[93] Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. BitBlaze: A new approach to computer security via binary analysis. In *Information systems security*, pages 1–25. Springer, 2008.

[94] Kapil Anand, Matthew Smithson, Aparna Kotha, Khaled Elwazeer, and Rajeev Barua. Decompilation to compiler high IR in a binary rewriter.

[95] Chris Eagle. *The IDA pro book: the unofficial guide to the world's most popular disassembler.* No Starch Press, 2011.

[96] DWARF Standards Committee et al. The dwarf debugging standard, 2008.

[97] Tien-Fu Chen and Jean-Loup Baer. *Reducing memory latency via non-blocking and prefetching caches*, volume 27. ACM, 1992.

[98] David Callahan, Ken Kennedy, and Allan Porterfield. Software prefetching. In *ACM SIGARCH Computer Architecture News*, volume 19, pages 40–52. ACM, 1991.

[99] Sam Ainsworth and Timothy M. Jones. Software prefetching for indirect memory accesses. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization*, CGO '17, pages 305–317, Piscataway, NJ, USA, 2017. IEEE Press.

[100] David H Bailey, Eric Barszcz, John T Barton, David S Browning, Robert L Carter, Leonardo Dagum, Rod A Fatoohi, Paul O Frederickson, Thomas A Lasinski, Rob S Schreiber, et al. The NAS parallel benchmarks summary and preliminary results. In *Supercomputing, 1991. Supercomputing'91. Proceedings of the 1991 ACM/IEEE Conference on*, pages 158–165. IEEE, 1991.

[101] Cagri Balkesen, Jens Teubner, Gustavo Alonso, and M Tamer Özsu. Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware. In *Data Engineering (ICDE), 2013 IEEE 29th International Conference on*, pages 362–373. IEEE, 2013.

[102] TrailofBits. Translating x86 binaries to LLVM IR. `https://github.com/trailofbits/mcsema`, 2014.

[103] Saeed Maleki, Yaoqing Gao, Maria J Garzar, Tommy Wong, David A Padua, et al. An evaluation of vectorizing compilers. In *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*, pages 372–382. IEEE, 2011.

[104] Kemal Ebcioglu, Erik Altman, Michael Gschwind, and Sumedh Sathaye. Dynamic binary translation and optimization. *IEEE Transactions on Computers*, 50(6):529–548, 2001.

[105] Amitabh Srivastava, Andrew Edwards, and Hoi Vo. Vulcan: Binary transformation in a distributed environment. Technical report, technical report msr-tr-2001-50, microsoft research, 2001.

[106] Giuseppe Desoli, Nikolay Mateev, Evelyn Duesterwald, Paolo Faraboschi, and Joseph A Fisher. DELI: A new run-time control point. In *Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture*, pages 257–268. IEEE Computer Society Press, 2002.

[107] Markus Mock, Craig Chambers, and Susan J Eggers. Calpa: a tool for automating selective dynamic compilation. In *Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, pages 291–302. ACM, 2000.

[108] Brian Grant, Markus Mock, Matthai Philipose, Craig Chambers, and Susan J Eggers. DyC: an expressive annotation-directed dynamic compiler for c. *Theoretical Computer Science*, 248(1-2):147–199, 2000.

[109] Sheldon Lobo. The Sun Studio binary code optimizer. `http://www.oracle.com/technetwork/server-storage/solaris/binopt-136601.html`, 1999.

[110] Microsoft. Binary profile feedback optimization framework. `https://www.microsoft.com/windows/cse/bit_projects.mspx`, 2005.

[111] Michael J Voss and Rudolf Eigemann. High-level adaptive program optimization with ADAPT. In *ACM SIGPLAN Notices*, volume 36, pages 93–102. ACM, 2001.

[112] Niall Murphy. Discovering and exploiting parallelism in DOACROSS loops. Technical report, University of Cambridge, Computer Laboratory, 2016.

[113] SPEC. CPU2006. `https://www.spec.org/cpu2006/`, 2006.

[114] James R Larus. Loop-level parallelism in numeric and symbolic programs. *Parallel and Distributed Systems, IEEE Transactions on*, 4(7):812–826, 1993.

[115] Minjang Kim, Hyesoon Kim, and Chi-Keung Luk. SD3: A scalable approach to dynamic data-dependence profiling. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 535–546. IEEE Computer Society, 2010.

[116] J Gregory Steffan and Todd C Mowry. The potential for using thread-level data speculation to facilitate automatic parallelization. In *High-Performance Computer Architecture, 1998. Proceedings., 1998 Fourth International Symposium on*, pages 2–13. IEEE, 1998.

[117] David W Wall. *Limits of instruction-level parallelism*, volume 19. ACM, 1991.

[118] Monica S. Lam and Robert P. Wilson. Limits of control flow on parallelism. *SIGARCH Comput. Archit. News*, 20(2):46–57, April 1992.

[119] Neil Vachharajani, Ram Rangan, Easwaran Raman, Matthew J Bridges, Guilherme Ottoni, and David I August. Speculative decoupled software pipelining. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, pages 49–59. IEEE Computer Society, 2007.

[120] Richard H Littin, JA David McWha, Murray W Pearson, and John G Cleary. Block based execution and task level parallelism.

[121] J.T. Oplinger, D.L. Heine, and M.S. Lam. In search of speculative thread-level parallelism. In *Parallel Architectures and Compilation Techniques, 1999. Proceedings. 1999 International Conference on*, pages 303–313, 1999.

[122] Jeffrey Oplinger, David Heine, Shih-Wei Liao, Basem A Nayfeh, Monica S Lam, and Kunle Olukotun. *Software and hardware for exploiting speculative parallelism with a multiprocessor.* Citeseer, 1997.

[123] Pedro Marcuello and Antonio González. A quantitative assessment of thread-level speculation techniques. In *Parallel and Distributed Processing Symposium, 2000. IPDPS 2000. Proceedings. 14th International*, pages 595–601. IEEE, 2000.

[124] Barbara Kreaseck, Dean Tullsen, and Brad Calder. Limits of task-based parallelism in irregular applications. In *High Performance Computing*, pages 43–58. Springer, 2000.

[125] Jonathan Mak, Karl-Filip Faxén, Sverker Janson, and Alan Mycroft. Estimating and exploiting potential parallelism by source-level dependence profiling. In *Euro-Par 2010-Parallel Processing*, pages 26–37. Springer, 2010.

[126] Georgios Tournavitis and Björn Franke. Semi-automatic extraction and exploitation of hierarchical pipeline parallelism using profiling information. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT '10, pages 377–388, New York, NY, USA, 2010. ACM.

[127] Milind Kulkarni, Martin Burtscher, Rajeshkar Inkulu, Keshav Pingali, and Calin Casçaval. How much parallelism is there in irregular applications? In *ACM Sigplan Notices*, volume 44, pages 3–14. ACM, 2009.

[128] Tobias JK Edler von Koch and Björn Franke. Limits of region-based dynamic binary parallelization. In *ACM SIGPLAN Notices*, volume 48, pages 13–22. ACM, 2013.

[129] Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman. *Compilers, Principles, Techniques.* Addison wesley Boston, 1986.

[130] Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM journal on computing*, 1(2):146–160, 1972.

[131] Pascal Felber, Christof Fetzer, and Torvald Riegel. Dynamic performance tuning of word-based software transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 237–246. ACM, 2008.

[132] Aparna Kotha, Kapil Anand, Matthew Smithson, Greeshma Yellareddy, and Rajeev Barua. Automatic parallelization in a binary rewriter. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, pages 547–557, Washington, DC, USA, 2010. IEEE Computer Society.

[133] Efe Yardimci and Michael Franz. Dynamic parallelization and mapping of binary executables on hierarchical platforms. In *Proceedings of the 3rd conference on Computing frontiers*, CF '06, pages 127–138, New York, NY, USA, 2006. ACM.

[134] Ben Hertzberg. *Runtime Automatic Speculative Parallelization of Sequential Programs*. PhD thesis, Stanford University, 2009.

[135] Jing Yang, Kevin Skadron, Mary Lou Soffa, and Kamin Whitehouse. Potential of dynamic binary parallelization. In *Workshop on Unique Chips and Systems UCAS-7*, page 51, 2012.

[136] Louis-Noël Pouchet. Polybench: The polyhedral benchmark suite. *URL: http://www. cs. ucla. edu/pouchet/software/polybench*, 2012.

[137] GCC. Automatic parallelization in gcc. `https://gcc.gnu.org/wiki/AutoParInGCC`, 2012.

Appendix A

Installation and Running Instructions

A.1 GBR Installation

Firstly download the latest version of DynamoRIO. Put the DynamoRIO files in the "external" folder. Then modify the root `CMakeLists.txt`

```
set(DynamoRIO_DIR "${YOUR_PATH_TO_DYNAMORIO}/cmake")
```

GBR can be linked with all versions of DynamoRIO. However there is one exception for the GABP paralleliser. It only supports DynamoRIO version 5.0 downwards. This is because a fraction of GABP implementation relies on `pthread` for parallelisation. Private loaders in newer version of DynamoRIO prevent linking its client with `pthread` library on `linux`. For DynamoRIO version 5.0 downwards, `pthread` can be linked by disabling the flag "-no-private-loader". A permanent solution (custom thread library) is still being developed.
To build the GBR project, go to the directory of GBR:

```
mkdir build
cd build
cmake ..
make -j
```

After building, there are several components generated:

- `bin/analyse`: the static binary analyser
- `lib/libpft.so`: client library for software prefetcher
- `lib/libvct.so`: client library for automatic vectorisation
- `lib/libgabp.so`: client library for GABP
- `lib/libbeep.so`: client library for BEEP
- `lib/libplan.so`: client library for loop profiler (lightweight BEEP)
- `lib/libgltimer.so`: client library for the loop timer
- `lib/libgftimer.so`: client library for the function timer

There are a few convenient bash scripts in the gabp folder.

- `gabp/parallel`: run the static analyzer and invoke GABP parallelisation
- `gabp/model`: run the static analyzer and invoke BEEP profiling
- `gabp/profile`: run the static analyser, run the BEEP profiler and run the static analyser again
- `gabp/time`: run the static analyzer and call the loop coverage profiling
- `gabp/ftime`: run the static analyzer and call the function coverage timer
- `gabp/graph`: run the static analyzer and generate CFG graph of the loop as pdf.

A.2 How to Run

If you wish to skip the trouble of loop selection, just run the existing tests.

```
cd test/spec2006/integer/462.libquantum
#run sequential unmodified binary
time ./libquantum_base.amd64-gcc-O2 1397 8
#run the paralleliser with four threads
../../../../gabp/parallel libquantum_base.amd64-gcc-O2 <thread_count> 1397 8
#see the timing difference in your machine
```

A.3 Standard Flow for Automatic Parallelisation

static binary analyser

We perform static binary analysis and generate hint files to guide the binary translation. If you run the static binary analyser:

```
/bin/analyze -p bzip2_gcc_O3
```

A static hint program `bzip2_gcc_O3.hint` is generated. This hint program is obfuscated but you can examine the contents using the `hintdump` tool.

```
/bin/hintdump bzip2_gcc_O3.hint
```

Step 0: go to the directory of your executable

In the repository, the “gabp” folder contains all necessary scripts. Assume the relative path to the gabp folder in the repository is

```
GABPTOOL=../../gabp/
GABPBIN=../../bin/
```

Assume the executable is `foo` and its arguments are `foo_arg`.

Step 1: loop coverage profiling

Firstly GABP needs to profile the binary to find the most profitable loop to parallelise. Through profiling, it can find “hot” loops with high timing coverage. The coverage profiling needs to be done one loop per run due for accuracy, therefore it probably takes very long time for executables with thousands of loops. For small executables, you can use the loop timer:

```
$GABPTOOL/time foo foo_arg
```

For big executables with thousands of loops:

```
$GABPTOOL/timeParallel foo foo_arg
```

The script uses the GNU `parallel` script to run each loop timer concurrently. Make sure you have the resources to compute. It puts the profiled coverage in a csv file called `foo.loopcov.csv`.

For complex executable with recursive calls, loop timer might not be fully accurate, you might also need to run a function coverage profiling to make sure the loop timer is consistent with the parent function:

```
$GABPTOOL/ftime foo foo_arg
```

The result of the function coverage is in `foo.funccov.csv`.

Step 2: data dependence profiling for high coverage loop

From the `foo.loopcov.csv` you can see the loop id, parent function name, start instruction id, and tool invocation counts for each loop. Currently there is not a fixed script to automatically filter the loops with certain filter threshold. For the moment you have to filter the loop manually with your own conditions. In the future, the automatic script will be implemented. Open the `foo.loopcov.csv`, rank them by time coverage, remove all loops with less than a threshold, delete all lines for small loop entries, save the file into another file `foo.loop.plan`.

Then run BEEP data dependence planner with the selected loop, you need to give an extra parameter for number of hypothetical threads for parallelisation and it generates a speedup estimation based on your core count. Note that this might also take a long time to run:

```
$GABPTOOL/plan foo <your_hypothetical_machine_core_count> foo_arg
```

The script will invoke BEEP profiling with specified loops. If you selected multiple loops and have computing resources, use the `planParallel` script:

```
$GABPTOOL/planParallel foo <your_hypothetical_machine_core_count> foo_arg
```

Step 2.1: change BEEP parallel execution model

The default execution model for BEEP is the ideal induction/reduction optimisation model. To change to other models, you can set the parameters as environment variables such as:

```
export EM_MODEL_TYPE=2
```

Or you can directly modify the file in `gabp/parameters`.

After profiling, BEEP summaries the dynamic data dependence graph into a `.ddg` file. This `.ddg` concludes the parallelism information for your selected loop. It will be loaded into static binary analyser for generating parallelisation hint programs. You can visualise each ddg using GABP utility binary called `rddg`:

```
$GABPBIN/rddg Loop_{id}.ddg
#convert to a dot graphviz file
$GABPBIN/rddg -d Loop_{id}.ddg
```

Step 3: loop selection

The automatic loop selection is still being implemented, for the moment you need to do it manually by checking each ddg and reason about the dependences.

```
#analyse the foo executable
$GABPBIN/analyse -a foo
#visualise loop files
$GABPTOOL/graph foo.loop
#find it in the pdf
evince foo.loop.pdf &
```

Locate the CFG for an investigated loop and print the DDG from the `loop.ddg` file. If all the cross-iteration dependences are only induction variables, then it is a DOALL loop, otherwise it is a normal loop. You can also run execution model to further verify the estimated speedup for each loop under different parallelisation models but it is optional.

Put your selection of loops in the `foo.loop.select` file with the following format:

```
<loop_id> <loop_type>
```

where the `loop_type` can be found in `Loop.h`.

Step 4: automatic parallelisation

Just run the paralleliser without further manual intervention:

```
$GABPTOOL/parallel foo <num_threads> foo_arg
```

Appendix B

Hint Instruction Set Architecture

Hint Opcode	Decription
	Application Modification Hints
APP_SPLIT_BLOCK	Inserts a jump to the next PC so that it splits the current basic block.
APP_INSERT_JUMP	Inserts a jump to a specified address and terminates the current basic block
APP_INSERT_CALL	Inserts a call to a specified address
APP_GEN_CODE	Calls JIT code generation from specified DynamoRIO IR
APP_REPLICATE_CODE	Replace the dynamic code with specified code snippet.
APP_GEN_STM	JIT generate STM code main routines.
APP_GEN_SYNC	JIT generate synchronisation main routines.
APP_ALLOC_STACK	Allocate a new stack for current execution and switch to the stack.
APP_SWITCH_STACK	Switch to the specified stack.
APP_INC_STACK	Increment stack pointer by specifed offset.
APP_DEC_STACK	Decrement stack pointer by specifed offset.
APP_SAVE_REG	Spills the specified registers mask to thread local storage including eflags
APP_RESTORE_REG	Spills the specified registers mask to thread local storage including eflags
APP_DELETE_INSTR	Delete specified instructions from current address with specified range.
OPT_PREFETCH	Perform a memory prefetch with specified address and offset.
	Automatic Parallelisation Hints
PARA_THREAD_CREATE	Create specified number of threads and place them in thread pool.
PARA_THREAD_DELETE	Delete threads from the thread pool.
PARA_SCHED_THREAD	Schedule threads to jump to specified loop code.
PARA_YIELD_THREAD	Force the current thread to jump back to thread pool.
PARA_LOOP_INIT	Annotates the init block of a loop.
PARA_LOOP_ITER	Annotates the start block (start of iteration) of a loop.
PARA_LOOP_EXIT	Annotates the exit block of a loop.
PARA_CALL_START	Annotates the start of subroutine (call instruction) of a loop.

PARA_CALL_END	Annotates the end of subroutine (next PC of the call instruction) of a loop.
PARA_FUNC_HEAD	Annotates the head of the subroutine code.
PARA_FUNC_RETURN	Annotates the return of the subroutine code.
PARA_UPDATE_VAR	Update the loop's variable with specified value or loading address.
PARA_UPDATE_INDUCTVAR	Update the loop's induction variable with new update method.
PARA_UPDATE_CHECK	Update the loop's check conditions with specified condition.
PARA_PRIVATISE_ADDR	Allocate a thread private copy of the marked memory address. Rewrite the current address to the privatised location.
PARA_PRIVATISE_VAR	Allocate a thread private copy of the marked loop variable. Rewrite the variable address to the privatised location.
PARA_LOCK_ADDR	Inline a spin lock on the specified address. When the address is locked, it prevents other thread from accessing this address.
PARA_UNLOCK_ADDR	Inline the unlock code on the specified address. When the address is unlocked, other thread can try to acquire the lock.
PARA_PRODUCE	Copy specified data to a specified memory region. If the memory region is not consumed, it spins until the memory is consumed. It copies the data to the memory region.
PARA_CONSUME	Load data from the specified memory region. If the memory region is empty, it waits until the memory region is not empty. Once it consumes the memory region, it marks the region as consumed.
PARA_SIGNAL	Samed as PARA_PRODUCE but with JIT communication channels.
PARA_WAIT	Samed as PARA_CONSUME but with JIT communication channels.
PARA_SEQ_SEGMENT	Marks a region of code as sequential segments.
TRANS_START	Start a transaction by check-pointing current register states
TRANS_COMMIT	Finish a transaction by validating read sets and commiting write sets. If validation fails, revert to the PC recorded by the check point.
SPEC_MEM_ACCESS	Redirect current memory access to the transaction's read or write set using hash tables.
SPEC_MEM_FAST	Encode one read or write direct entry address to current instruction.
SPEC_READ	Speculative read on the memory load.
SPEC_WRITE	Speculative write on the memory write.
SPEC_REG_ACCESS	Redirect current memory access to the transaction's register read or register write set.
TRANS_WAIT	Wait until it is the oldest thread.
RDTSC_START	Record the current time stamp.
RDTSC_END	Record the current time stamp and perform subtraction on the previous RDTSC_START timestamp.
PRF_START	Start the profiler at given address.
PRF_END	Turn off the profiler at given address.
PRF_LOOP_START	Annotate the start of the loop for profiling.

PRF_LOOP_ITER	Annotate the start of the loop iteration for profiling.
PRF_CALL_START	Annotate the start of the function call for profiling.
PRF_CALL_END	Annotate the end of the function call for profiling.
PRF_SEEN_BLOCK	The basic block is covered at runtime.
BEEP_STATIC_UNDECIDED	Annotate the current memory address that can not be decided by static binary analysis.
GABP_DEBUG	Insert a interrupt at current address.

Table B.1: Hint Instruction Opcode