# PROGRAM ORIENTEERING

BY

NAVEEN NEELAKANTAM

B.S., University of Illinois at Urbana-Champaign, 2001

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2004

Urbana, Illinois

# TABLE OF CONTENTS

# 1 INTRODUCTION

A general property of computer programs is commonly referred to as the 90/10 rule: a program spends approximately 90% of its execution time in approximately 10% of the (executed) static program. The 90/10 rule intuitively derives from programs spending most of their execution time inside of loops (or mutually recursive functions). We will refer to the frequently executed 10% of the static program as *hot* and the remaining 90% as *cold*.

Computer system designers frequently exploit the 90/10 property of programs in order to focus their optimization efforts. A significant reduction in the execution time of the hot 10% of a program will result in a significant reduction in the overall running time.[1] In particular, optimization techniques using feedback-directed optimization (FDO) attempt to gather information about hot parts of a program in order to guide optimization.

Figure 1.1 shows the dynamic execution count for the static instructions in `vortex`.[2] As expected `vortex` displays the 90/10 property: 15% of the (executed) static instructions are responsible for 90% of the overall execution time. The figure shows that the majority of static instructions are seldom or never executed, and the hot instructions tend to be clustered in contiguous regions.

The focus of this work is a hardware and software mechanism that can identify these hot program regions. The mechanism is part of a processor framework capable of applying compiler transformations. The novel aspect of the proposed mechanism is its ability to accurately characterize the hot program regions such that a runtime compiler can beneficially apply speculative code optimizations.

The remainder of this section is organized as follows. Section 1.1 begins by discussing prior optimization frameworks. Section 1.2 overviews the proposed mechanism and the framework used in this work. Section 1.3 highlights the contributions of our mechanism and outlines the remainder of this work.

## 1.1 Feedback-Directed Optimization Frameworks

Feedback-directed optimization (FDO) frameworks collect profile information (feedback) from the execution of a program and use it to guide optimization. The profile helps characterize the behavior of the program, and an optimizing compiler then uses this information to optimize the program for common behavior.

[1]Clearly Amdahl's law implies that the cold part of a program will increasingly contribute to the overall execution time as the hot parts are optimized. This does not invalidate the 90/10 rule, but it implies that the 10% shifts as optimizations are applied.

[2]The results are from the entire execution of `vortex` using a reduced reference input set. We do not present similar figures for the other SPECint2000 benchmarks, but they display similar characteristics.

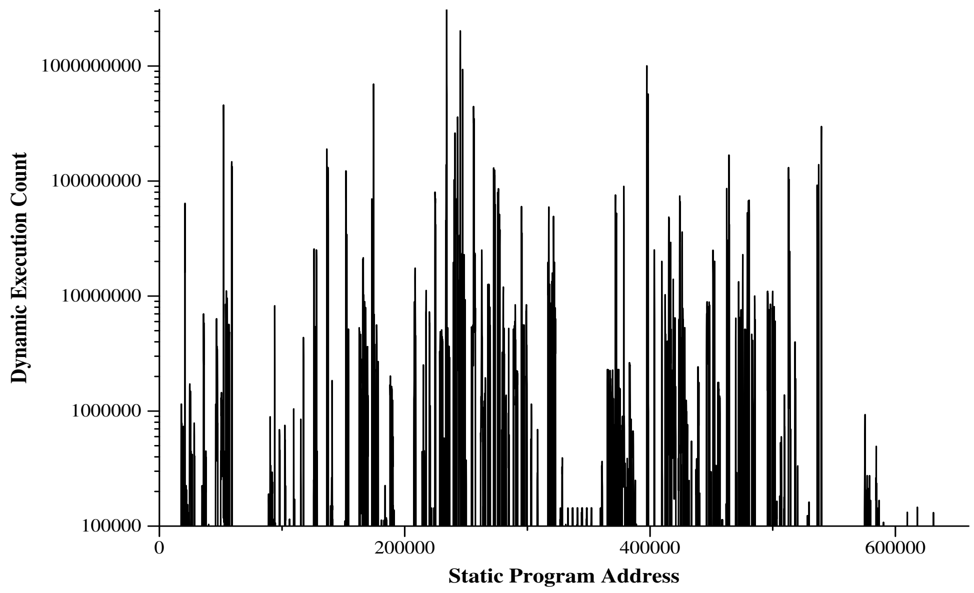

**Figure 1.1: Dynamic execution count of static instructions in vortex**

Two general classes of feedback-directed optimization frameworks exist: static and dynamic. Static FDO frameworks (e.g., FX!32 [1] and Spike [2]) collect profiles from a set of program executions and recompile and optimize the program offline. The optimized program can then replace the original during subsequent executions. Because static FDO frameworks perform optimization offline, resource-intensive techniques are viable and a large amount of optimization effort can be invested. However, the optimization potential of information gathered during an execution run of the program cannot be exploited until after that run has completed. This introduces the possibility that the collected profile will incorrectly characterize future runs, and, as a result, the optimizations based on the profile may not improve, or, could even degrade, performance.

In contrast, dynamic FDO frameworks (e.g., CMS [3], DAISY [4], Dynamo [5], IA32EL [6] and Jalapeño [7]) collect profile feedback and employ optimizations during execution of the program. The dynamic framework periodically analyzes the profile information collected and applies optimizations suggested by recent program behavior. As a result, a dynamic framework can potentially improve program performance during an execution, but must also account for any profiling and recompilation overhead. To achieve a net performance improvement, the benefits of optimization must exceed the cost of profiling and recompilation.

Because of the need to amortize overhead, some FDO frameworks employ staged optimization techniques. All regions are initially optimized using lightweight techniques or left unoptimized, in order to reduce overhead. As regions are executed more frequently, they pass through additional optimization stages that apply progressively higher-levels of optimization (eventually approaching levels used by static FDO frameworks). As a result, staged dynamic FDO frameworks such as Dynamo [5] and Jalapeño [7]) balance the cost of optimization effort against the expected performance gain for a given region.

Even so, there are limits to the degree of optimization that can practically be achieved by FDO frameworks for traditional architectures. The collected profile merely indicates past program behavior, and it is therefore difficult for the FDO framework to distinguish code that has merely not *yet* been needed from code that will never be needed. As a result, such systems conservatively retain all of the code in the original program, and only optimize it such that the expected behaviors are made fast at the expense of the unexpected ones.

Master/Slave Speculative Parallelization (MSSP) is a novel processor paradigm that overcomes this limitation. MSSP enables an FDO framework to generate an optimized version of the program that matches expected behavior without retaining the code necessary for the unexpected behavior [8, 9]. An FDO framework for MSSP generates an optimized version of the program, called the *approximate* program, using *approximation* transformations that are allowed to violate correctness requirements. An MSSP-enabled processor uses the approximate code in conjunction with the original code to provide the full functionality of the original program at the speed of the approximate program. See Appendix A for a brief overview of the MSSP paradigm and the performance benefits of approximate code.

Section 1.2 describes a staged optimization framework for a processor implementing the MSSP paradigm. MSSP provides our framework with a wide range of optimization flexibility in generating the approximate program. However, many of the potential optimizations are highly speculative, and our framework must have an accurate characterization of program behavior before applying them.

## 1.2 Overview of the Program Orienteering Framework

Any dynamic FDO framework must learn about the program being executed before it can beneficially apply optimizations. Initially, a dynamic optimizer will have no information and must characterize the running program, a process that we refer to as *program orienteering*.[3] For this purpose, we propose a hardware-software mechanism that we call a *program orienteer* to ascertain enough information to direct generation of approximate code. The program orienteer has the following requirements:

1. **The program orienteer is able to identify hot program regions.** This is of the utmost importance to any dynamic optimizer. The frequently executed regions of the program should be optimized to maximize performance.

2. **Hot program regions are identified and characterized in parallel with full-speed native execution.** Execution of the original program does not need to pause while hot regions are characterized. This avoids the overhead that is typical of other dynamic optimization frameworks.

3. **Hot program regions are targeted for additional profiling to expedite learning about the program's behavior.** The orienteer inserts software profiling instructions into approximate code that is otherwise identical to the original hot code. This profiled code is used to gather information that can accurately guide staged optimization.

The program orienteer uses both hardware and software mechanisms to meet these requirements. It uses hardware to perform the simple, repetitive tasks of collecting profile information for the program; a relatively small amount of hardware would be needed, and hardware provides a low-overhead method of accomplishing the task. It uses software to

[3]Orienteering is a competitive sport in which participants (orienteers) use a map and compass to navigate through unfamiliar landscape and find specific goal points

provide sophisticated program analysis, which can be decoupled from native execution in a multiprocessor system.

Figure 1.2 abstractly depicts our dynamic FDO framework and how it interacts with the program orienteer. It is composed of a processing unit (which is an abstraction for an actual processor architecture), program orienteer software, a runtime compiler, two data structures stored in memory (the profile data and the Intermediate Representation (IR)), and two static binaries (the original program and the approximate program).

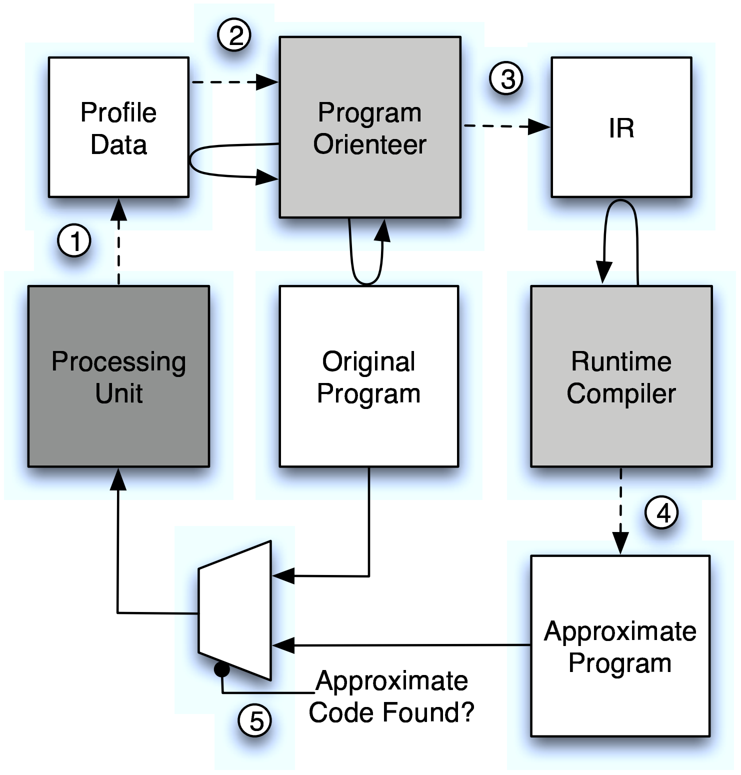

**Figure 1.2: Diagram of the program orienteering framework.**

The role of the processing unit is to execute the original program or its corresponding approximate-program region if one exists. While executing the original program, the processing unit samples the dynamic targets of control-flow instructions with an augmented hardware profiler and stores the edge samples into the profile data structure ①.

Periodically the profile samples are inspected to find *hot loop seeds* (described in Section 2.1), which are provided to the program orienteer as starting points ②. The program orienteer software begins exploring the original program from these starting points and uses profile data and the original-program binary as guides. As the program orienteer explores the program it constructs a compiler Intermediate Representation (IR) ③ that contains higher-level structures (i.e., basic blocks and functions) and the control relationships between them (i.e., a Control-Flow Graph (CFG) and a call graph).

After a region has been explored, the runtime compiler uses the information in the IR to generate approximate code ④, and the framework stores a mapping between the original code and the new approximate code region. As the processing unit executes the original program, the framework searches for corresponding mappings, and, if it finds one, passes the processing unit the mapped approximate code for execution ⑤.

The first time a region is approximated, the orienteer inserts profile instructions into it. When executed, these instructions collect additional profile data that provides the program orienteer with more accurate information about program behavior. Gradually the orienteer will generate a characterization of the program that can be used to apply speculative compiler optimizations.

## 1.3 Contributions

This work differs from previous research on FDO frameworks in three ways. First, our framework executes the original program binary in hardware, obviating the need for software interpretation. Second, it does not depend entirely upon an initial profile of a program region to guide optimization, but, instead, utilizes flexible profile instructions to refine program characterization. Third, our framework carefully reconstructs a meaningful IR of the dynamic program.

As described in Section 2.3, our framework executes the original program in hardware rather than interpreting it in software. As a result, our framework is not rushed to generate a natively executable binary in order to circumvent the large performance penalties associated with software interpretation. However, we do not have the privilege of collecting profile information while interpreting, so we have to use a separate mechanism. For this work we augment the processor that executes the original program with a hardware profiler that samples control flow.

Because our hardware-collected profile is sampled, our framework presumes that the initial profile suffers from sampling error. We therefore use the initial profile to generate conservative approximate code that we instrument with profile instructions. The profile instructions provide more accurate information that guides speculative approximation. In Section 2.2.2 we describe how our framework uses profile instructions to speculatively remove highly biased branches.

Although this work does not fully implement a runtime compiler, we believe that the success of approximation and optimization passes will depend on the orienteer's ability to generate a meaningful compiler intermediate representation. Because the original-program binary does not retain high-level information, the orienteer takes great care when constructing an IR for the program, especially when considering optimized original program binaries. Chapter 3 provides illustrating examples.

In Chapter 2 we will describe the design of our dynamic FDO framework in greater detail. Chapter 3 will provide code examples that illustrate some of the issues confronting the orienteer. Chapter 4 will present experimental results and we will conclude in Chapter 5.

# 2 SYSTEM DESIGN

This section expands upon the staged dynamic FDO framework introduced in Section 1.2 and depicted in Figure 1.2. Specifically, it elaborates upon the design of the program orienteer. Section 2.1 provides an overview of the program orienteer. Section 2.2 describes how the program orienteer can aid generation of approximate code. Section 2.3 briefly describes the hardware components in our framework.

The framework has been designed specifically for a processor implementing the MSSP paradigm. The MSSP paradigm is particularly amenable to dynamic FDO techniques because of its ability to utilize an optimized version of an original program called the *approximate* program. A discussion of the MSSP paradigm and the approximate program can be found in Appendix A.

## 2.1 The Program Orienteer

Our framework invokes the program orienteer whenever a frequently executed backedge, called a *hot loop seed*, is discovered. The hot loop seed provides the program orienteer with a hint as to the location of a hot program loop. The target of the backedge is the address of the loop header, and the source of the backedge is the address of the loopback branch.

The program orienteer's algorithm for exploring hot program loops is shown in Figure 2.1. Exploration begins in function `ExploreHotLoop` which takes the source and target of the loop backedge as arguments (`hot_loop_tail` and `hot_loop_header` respectively). `ExploreHotLoop` starts by exploring the basic block at the target of the backedge by calling `ExploreBasicBlock`. It then follows Control-Flow Graph (CFG) edges out of each basic block in the worklist by calling `AddBasicBlockSuccessors`. Finally, it either declares the exploration process as successful if the loop backedge was explored or returns an error.

`ExploreBasicBlock` decodes the static instructions at the specified address and adds them to the basic block being explored one instruction at a time.[1] The process continues until a control instruction is found, which completes the basic block. The orienteer places the finished basic block onto the global worklist and marks the basic block address as having been explored.

`AddBasicBlockSuccessors` queries the collected profile data using the address of the specified basic block's terminating instruction. The function checks each profiled target of the control instruction to see if it exceeds a bias threshold. If the target is sufficiently biased, the orienteer will consider the path *active* and will explore the target basic block by calling `ExploreBasicBlock`. Otherwise, a special block, called a *stub* block, is added that

[1] The orienteer decodes the instructions by consulting the original-program text, which we presume is accessible in memory.

```
ExploreHotLoop(hot_loop_header, hot_loop_tail)
   ExploreBasicBlock(hot_loop_header)
   while worklist is not empty
      basic_block = block popped from worklist
      AddBasicBlockSuccessors(basic_block)
   //Return success if hot loop backedge was explored.
   if control-flow edge exists from hot_loop_tail to hot_loop_header
      return success
   else
      return error

ExploreBasicBlock(address)
   //Do not explore an address more than once.
   if address has already been explored
      return
   new basic_block
   done = false
   temp_addr = address
   //Add static instructions to basic_block until a
   // control instruction is found.
   while not done
      add static instruction at temp_addr to basic_block
      if static instruction at temp_addr is a control instruction
         done = true
      increment temp_addr
   mark address as having been explored
   push basic_block onto worklist

AddBasicBlockSuccessors(basic_block)
   control_inst = terminating instruction of basic_block
   profiled_targets = list of profiled target addresses for control_inst
   //The fallthrough path of a direct control instruction
   //will be included in the profiled targets
   for each target in profiled_targets
      if target is sufficiently biased
         ExploreBasicBlock(target)
      else
         //A stub block denotes an unexplored path.
         create a stub block for target
      //Add a CFG edge for each profiled target
      add control-flow edge from control_inst address to target address
```

**Figure 2.1: The program orienteer loop exploration algorithm.**

denotes a path that has been skipped. The significance of the stub block will be discussed in Section 2.1.1.

The algorithm presented in Figure 2.1 does not address many of the more complicated cases common in real programs. Chapter 3 discusses some of these complexities and discusses how they are addressed by the program orienteer. However, the algorithm covers the basics and is sufficient for the example presented in Section 2.1.1.

### 2.1.1 Simple loop example

Figure 2.2 provides an example of the program orienteer in action over a simple hot loop. The figure depicts the various phases of the loop exploration algorithm, beginning with the hot loop seed. As shown in the figure, the result is a complete representation of the hot loop.

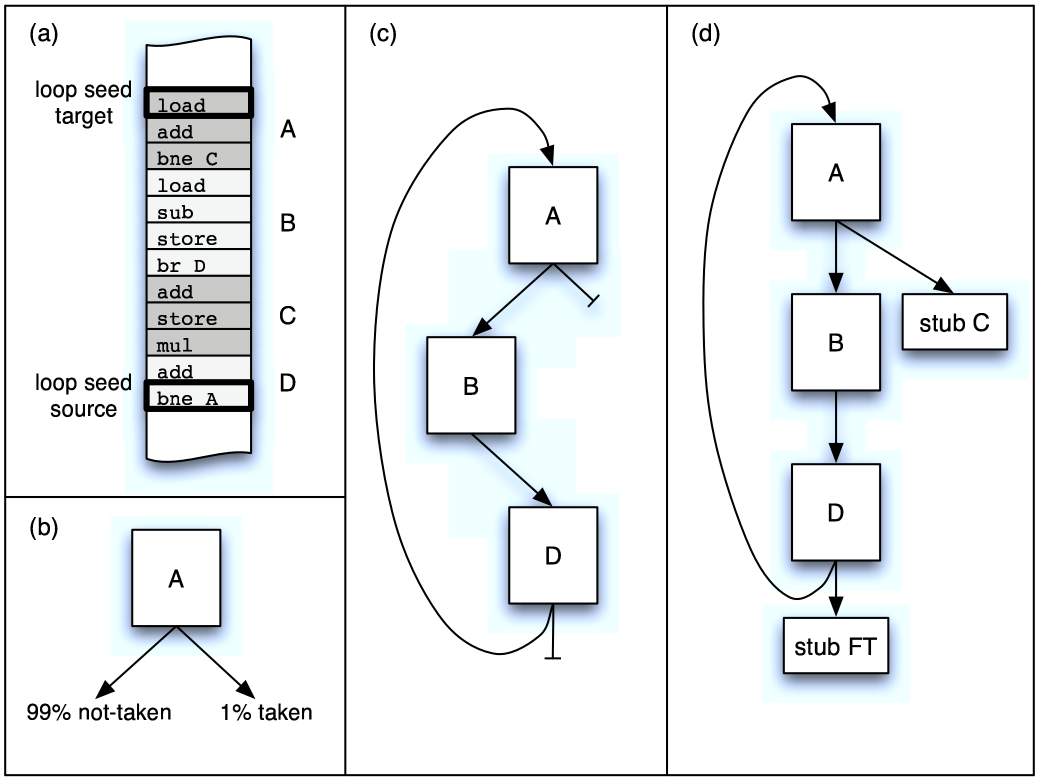

**Figure 2.2: Example of loop exploration by program orienteer.** (a) Original-program instructions with the loop seed denoted. (b) Exploration of the first basic block, and query results for the terminating instruction. (c) Exploration of the remaining active basic blocks and discovery of the loop seed source. (d) Insertion of stub blocks for omitted paths.

The orienteer provides the hot loop IR to a runtime compiler in order to generate an MSSP approximate code region. The runtime compiler will add the constructs necessary for the MSSP paradigm, namely the `entry`, `fork`, and `verify` blocks. The runtime compiler can also apply optimizations to the approximate code, but for simplicity we will not consider them at the moment.

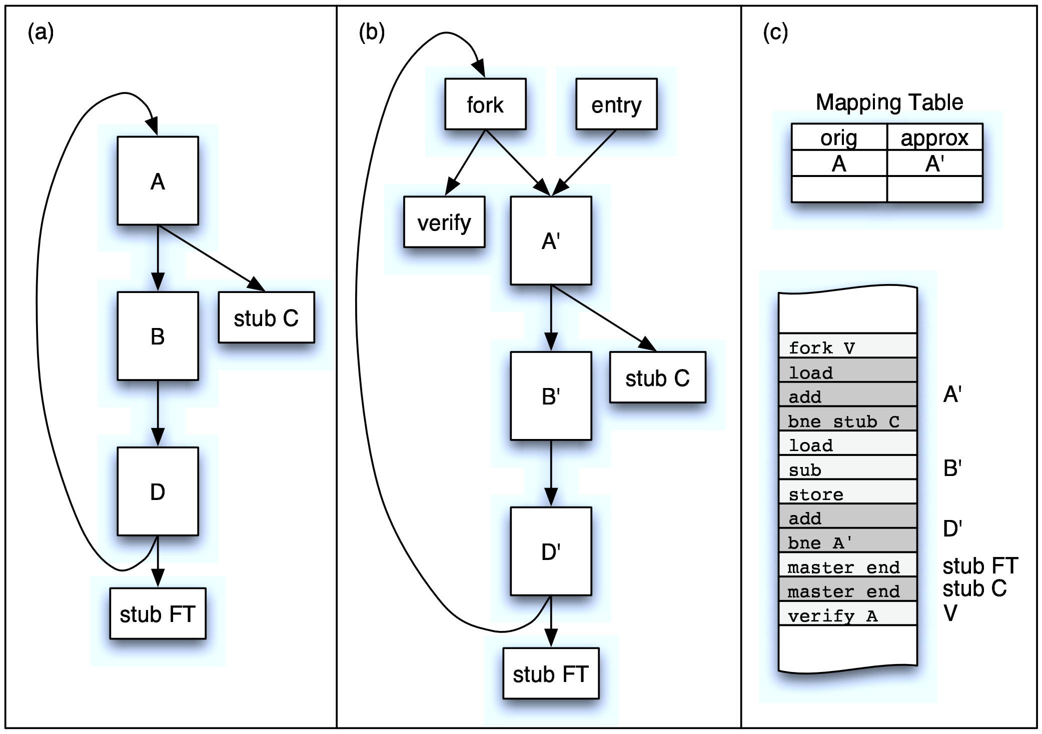

**Figure 2.3: Generation and deployment of approximate code region.** (a) Representation provided by orienteer (see Figure 2.2). (b) Corresponding approximate code representation. (c) Deployment of approximate code and update of MSSP mapping table.

Figure 2.3 depicts the layout and deployment of generated approximate code. The approximate code is written to available memory,[2] and the MSSP map table is updated. The map table associates the entry of the approximate code with an original-program PC so that execution can transition into MSSP mode (see Appendix A for details).

Note that the generated code for the stub blocks contains `master_end` instructions. The `master_end` instruction terminates the master processor in the MSSP paradigm. Essentially, this is a performance enhancement during program characterization and is discussed in Appendix A.

## 2.2 Refining the Approximate Code

The program orienteer's initial characterization of a region may either be slightly inaccurate (because the hardware edge profiler statistically samples control flow) or incorrect altogether (because program behavior changes). Alternatively, the orienteer may have generated overly conservative code and included unnecessary paths or instructions. To counteract either situation, the orienteer inserts software profiling instructions into the approximate code in order to gather more accurate information. The program orienteer uses the collected information to guide adaptation of the deployed approximate code as the characterization improves.

[2]For simplicity, presume that the system includes internal physical memory not visible to the operating system.

### 2.2.1 Software profiling

One of the implicit advantages to generating specialized code dynamically is that it can utilize implementation-specific ISA features exposed by the hardware. In our framework, the ISA has been expanded to include a flexible software profiling instruction. This new instruction essentially exposes a mechanism (akin to the hardware edge profiler described in Section 2.3) that can record the execution frequency of profiling instructions. The profile instruction encodes the ID of a counter that should be incremented and a threshold value. When the threshold is exceeded, it can invoke a software handler (which in our case is the program orienteer). The profile instruction also encodes a sampling rate that causes counter updates only to occur when the instruction is sampled.[3] The semantics of the profile instruction are even flexible enough to implement other profiling techniques such as value profiling or path profiling, but the use of these features is left for future work.

The program orienteer inserts these profiling instructions into the approximate code in order to gather more information about program behavior. For instance, the program orienteer inserts profiling instructions along paths in the approximate code region in order to determine if a path was correctly included or if a path is uncommon enough to be stubbed.

### 2.2.2 Stub adaptation

Our implementation of the orienteer instruments any stubs deployed in the approximate program by inserting profile instructions. Figure 2.4 depicts how profile instructions are used for this purpose. A pair of profile instructions are inserted into the stub block and its immediate predecessor block, which enables the program orienteer to learn about the relative execution frequencies of the blocks.

The threshold values for each profile instruction can be set independently and their ratio defines a branch bias threshold. For instance, if the stub threshold is 1% of the predecessor threshold and the predecessor branches to the stub more than 1% of the time, the stub counter will trigger, which indicates an active stub for the program orienteer to explore. On the other hand, if the predecessor branches to the stub less than 1% of the time, the predecessor counter will trigger, which indicates a cold stub that the program orienteer can remove.

Figure 2.5 uses the running example to demonstrate the stub adaptation process in which the stub block C becomes active. The orienteer removes the stub block and the path from block A to block C in the original program is explored. In the example, a new basic block C′ is inserted that precedes the loop-back block D′.

The deployed approximate code is updated with the new basic block and CFG edges. To minimize overhead, rather than regenerating the entire region, the necessary updates are made to the deployed code in place. First, the new basic block C′ is written to the nearest available program memory. Then, the profile instruction in block A′ is converted to a `nop` and the target of the terminating branch instruction in block A′ is updated with the starting address of block C′. Finally, the memory occupied by the removed stub block is freed for later use.

If, on the other hand, the orienteer learns that the stub path is cold, the orienteer will remove the stub. After doing so, it will convert both the branch to the stub block and the profile instruction in block A′ to `nop`'s, and will free the memory occupied by the stub

[3]The profile instruction sampling rate applies on a per instruction basis, and is not a global rate. Therefore, sampled profile instructions are not as susceptible to sampling error.

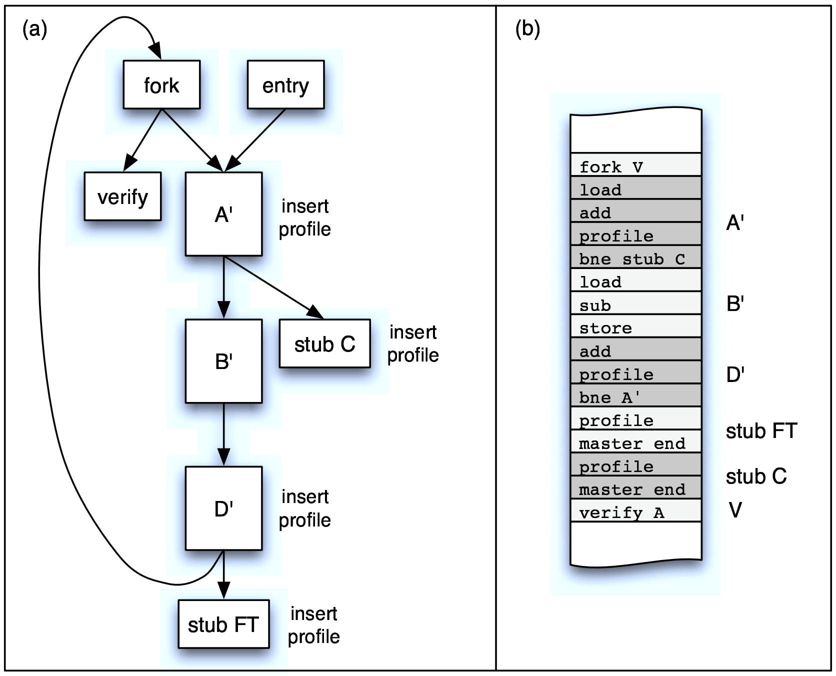

**Figure 2.4: Stub profiling.** (a) The orienteer inserts profiling instructions into all stub blocks as well as their predecessors. This enables the orienteer to learn about the execution frequency of each stub block relative to its predecessor. (b) The profiled approximate code.

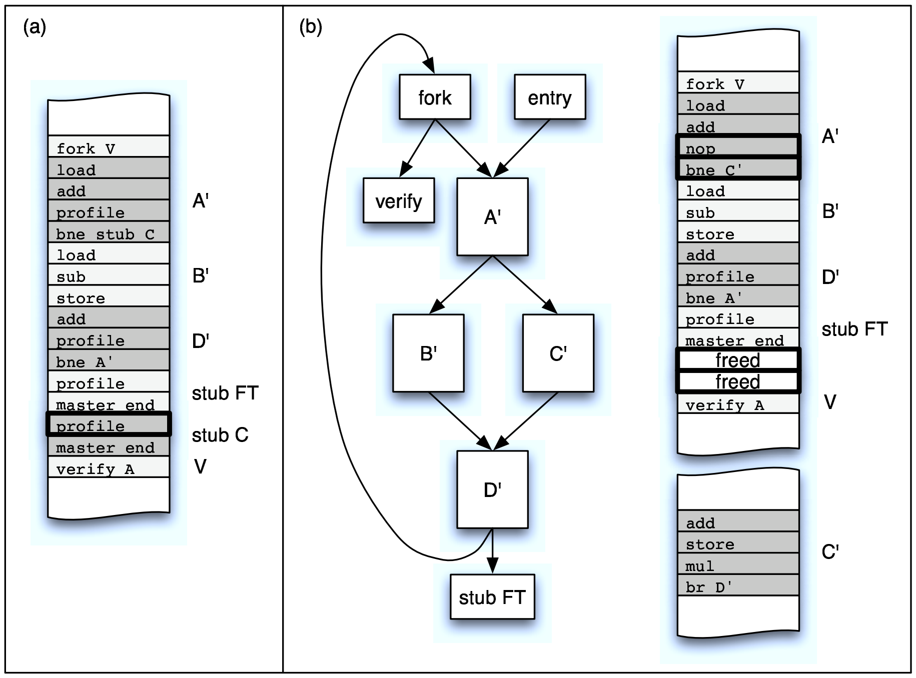

**Figure 2.5: Example of stub adaptation.** (a) The profile instruction in the stub block C overcomes its threshold. (b) In response the orienteer updates the approximate region by deploying the newly discovered basic block and updating instructions in block A′.

block. The removal of the branch in block A$'$ effectively merges A$'$ and B$'$ into a single basic block, and introduces additional optimization potential such as enabling the removal of branch computation code via dead-code elimination [10].

## 2.3 Hardware Components

Our framework targets a Chip Multiprocessor (CMP) that implements the MSSP execution paradigm. The CMP is asymmetrical and is composed of many small slave cores and a single large master core. In the studied implementation all cores execute the same ISA, but the master additionally supports the profile instructions that are used by the program orienteer to instrument the approximate code (Section 2.2.1) and the instructions necessary for the MSSP paradigm (see Appendix A). The slave cores execute the original-program binary when a corresponding approximate program region does not exist. In addition, the slave cores verify execution of the approximate program (as specified by the MSSP paradigm) and run the software components of the dynamic optimization framework (the program orienteer and runtime compiler).

Each slave core in the CMP is augmented with a small hardware edge profiler. This hardware edge profiler samples control-flow instruction outcomes and stores the information in a central location for later retrieval by the Program Orienteer. It is not necessary for the edge profiler to sample every instruction so long as it can create an accurate characterization of program control-flow behavior. This research presumes the edge profiler statistically samples control-flow outcomes. The exact implementation details of a hardware edge profiler and profile storage are beyond the scope of this thesis, but examples of implementations can be found in [11, 12].

Our framework periodically examines the profile data collected by hardware to find hot loop seeds that exceed a threshold. As described in Section 2.1 these hot loop seeds are provided to the program orienteer as a starting point for exploration. The source and target of the hot loop seed provides the orienteer with reference points as to the boundaries of the loop region. Because any cyclic control flow needs at least one backedge we are assured to find all hot loops.

Chapter 3 expands upon the ideas introduced in this section by discussing how the program orienteer addresses more realistic program examples. Chapter 4 presents experimental results for our implemented framework.

# 3 ELABORATING EXAMPLES

This section offers additional insight into program orienteering with elaborating examples that illustrate some of the features and complexities of our dynamic optimization framework. In all cases, the orienteer strives to build a representation that accurately represents the high-level organization of the program.

## 3.1 False Loop Seeds

In Section 2.1 we explained that our framework identifies hot loops in the program by detecting hot loop seeds, which are backedges. While it is true that any loop must include a backedge, the converse is not necessarily true. Occasionally, seeds identified by the framework are not part of a loop at all. The backedge is simply an artifact of layout decisions made by the compiler.

Figure 3.1 demonstrates one example of this situation that we have encountered in our experiments. In the example control-flow fragment, separate exit paths in a function share an exit block terminated by a return instruction. In the generated code, the path A→C falls through to the return block, but the path B→C must branch to the block. Because the compiler laid out block B after block C, the B→C branch is a backedge

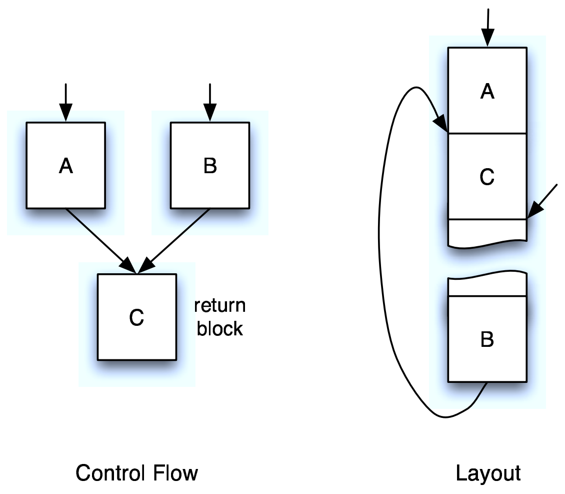

**Figure 3.1: Backedge that is not part of a loop.** Block C is the return block for a function and blocks A and B share the return path. Because block B is laid out after block C, a backedge is generated that is not part of a loop.

If enough samples are collected for the B→C backedge, it will be detected as a hot loop seed by our framework, and it will be passed to the program orienteer. The orienteer will start exploring at the target of the loop seed (block C), but will be unable to reach the source of the seed (block B) by following CFG edges. The program orienteer detects this situation and flags the seed as a false loop seed. These false positives can waste resources but will not lead to any incorrect conclusions.

## 3.2 Loops with Function Calls

Most hot program loops contain function calls, and our implementation of the program orienteer extends the algorithm described in Section 2.1 in order to handle them. Whenever the orienteer encounters a call instruction in the static program, it makes note of the call target and continues along the return (i.e., fallthrough) path. After the orienteer finishes exploring the control-flow in the loop, it recursively explores each call target and updates the call graph in our IR.

## 3.3 Tail-Call Elimination

If the last operation in a function is a call to another function, called a *tail call*, a compiler optimization known as tail-call elimination [10] can be applied. The optimization replaces the call and return instructions that would otherwise implement the function call with a branch instruction.

Tail-call elimination requires special consideration by the program orienteer. Because of the optimization, the program orienteer will not immediately recognize the tail-call branch as a function call. If we did not address this special case, the orienteer would find what appears to be a single function having two different entry points (see Figure 3.2). Instead, the orienteer should identify eliminated tail calls in order to develop a more accurate characterization of the original program.

Figure 3.2 illustrates a scenario where the orienteer identifies two eliminated tail calls. The program orienteer initially explores a function that contains the block $B_1$ and its successor $D_1$. At some later time, the orienteer explores a second function that also contains a block that has block $D_1$ as its successor. This violates a requirement of our IR that a basic block only be part of a single function, and it clues the orienteer in on the existence of eliminated tail calls.

In response, the orienteer splits the function containing $D_1$ into two separate functions. The orienteer represents the eliminated tail-call in the IR by inserting special tail-call blocks. A tail-call block replaces block $D_1$ as block $B_1$'s successor, and another tail-call block replaces block $D_1$ as block $A_2$'s successor. In addition, the orienteer places block $D_1$ into its own function and updates the call graph as if the tail calls were normal call paths. When the runtime compiler generates approximate code for the functions, it will reapply the tail-call elimination and implement the tail call as a branch.

## 3.4 Trimming Dangling Paths

As discussed in Section 2.1, the orienteer explores all paths that have enough samples to be considered active. In the case of a hot loop, several exit paths may be active and explored

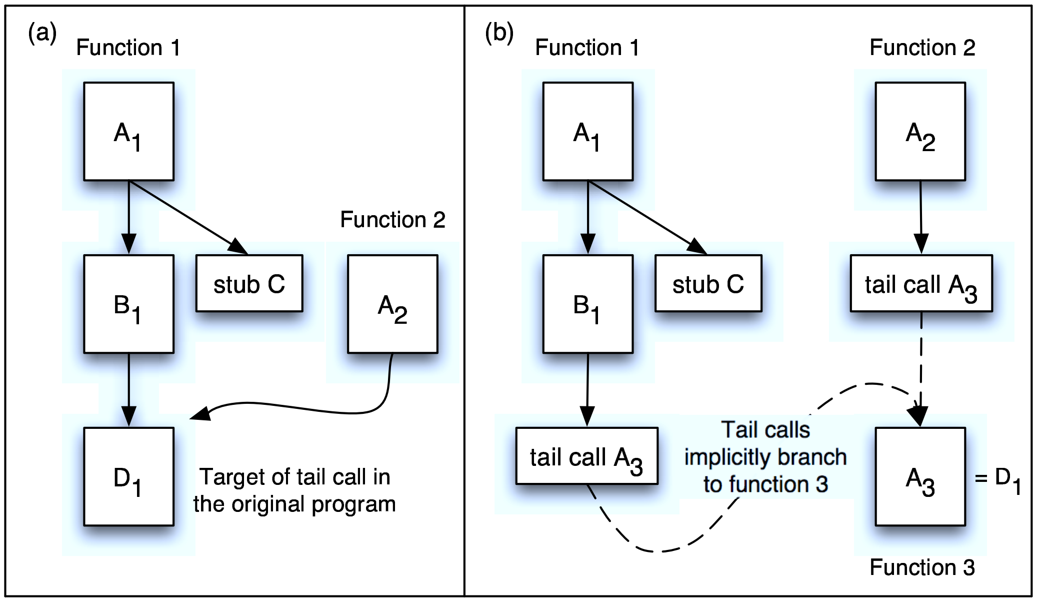

**Figure 3.2: Orienteering a tail call.** (a) The orienteer detects the presence of an eliminated tail call when it tries to add a CFG edge from block $A_2$ (in function 2) to block $D_1$ (in function 1). (b) The orienteer remedies the situation by moving block $D_1$ into its own function (logically renaming it to $A_3$) and inserting special tail-call blocks that denote the control edges from functions 1 and 2.

by the orienteer (see Figure 3.3). These *dangling paths* are not part of the characterized loop and will only be executed at most once per loop invocation (because they exit the loop). In most cases, only a few basic blocks are part of dangling paths, but occasionally larger dangling paths are found.

Rather than wasting available memory by approximating these paths, the orienteer removes them from the characterization of the loop region. The orienteer can easily identify and remove dangling paths. Any path that cannot reach the loop entry is not part of the loop being explored, and the orienteer removes blocks on any such path and replaces them with stub blocks. The resulting loop region will not contain any dangling paths (except for stub blocks, which implicitly dangle).

## 3.5 Nested Loops

Often a hot loop is contained within a hot outer loop. In general, the inner loop's backedge will execute much more frequently and will be detected as a hot loop seed first. The orienteer will explore the inner loop region and deploy approximate code for it. Later, the outer loop backedge may be detected and sent as a hot loop seed to the orienteer.

As shown in Figure 3.4, while exploring the outer loop, the orienteer will eventually encounter a CFG edge to a block in the inner loop region. The symptoms are similar to the tail-call elimination case described in Section 3.3, but differs because the CFG edge targets the loop header of a loop region rather than the middle of a function. The orienteer recognizes CFG edges that attempt to target the loop header of an explored loop, and will decide to subsume the inner loop's basic blocks into the outer loop region. In addition, all

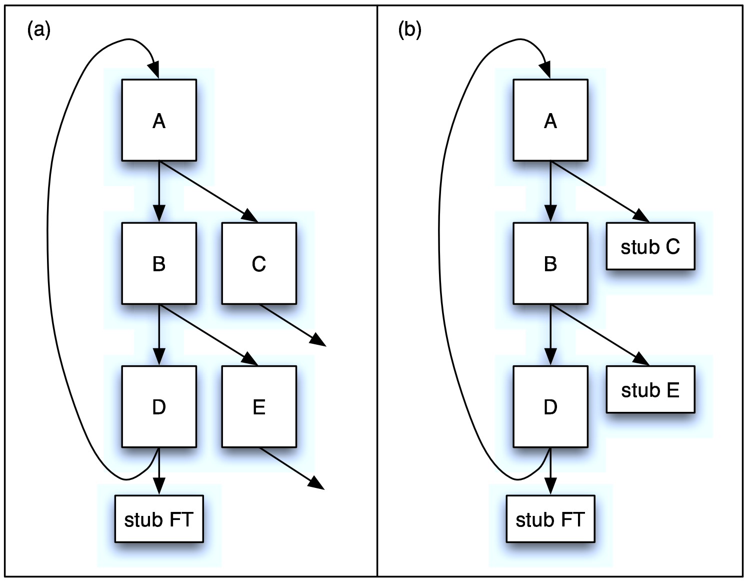

**Figure 3.3: Dangling path removal.** (a) The orienteer included blocks C and E in the loop region because the profile indicated that they are on active paths. However, neither block can reach the loop entry (block A) and therefore are on dangling paths. (b) The dangling blocks are replaced with stubs by the orienteer.

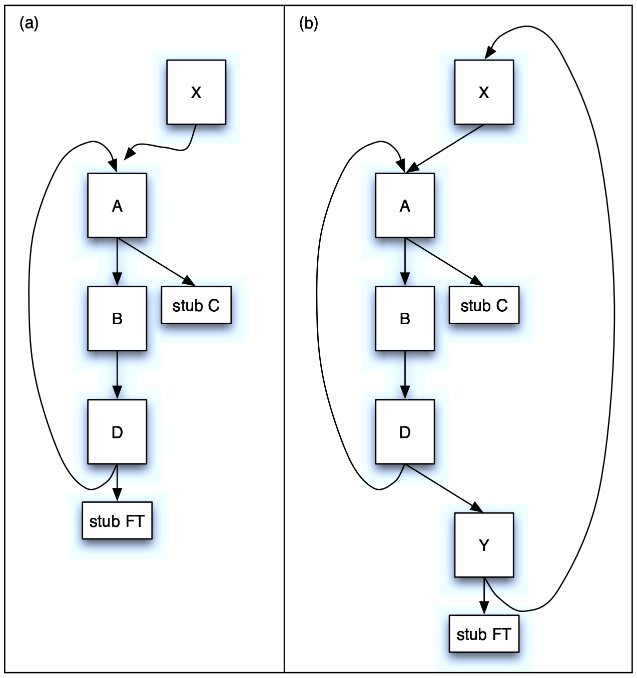

**Figure 3.4: Subsuming an inner loop.** (a) During exploration of a new region the orienteer encounters the loop header of an existing loop region. The orienteer subsumes the existing region into the new region and continues exploration at former stubs from the loop region. (b) The final IR after an outer loop region subsumes an inner loop region.

stub paths in the inner loop are reexamined, because the paths may need to be included in order to completely explore the outer loop region.

## 3.6 Indirect Branches

During exploration, the orienteer handles indirect branches similarly to how it handles conditional, direct branches. When the orienteer encounters an indirect branch, it explores any profiled targets with enough collected samples, and stubs the remaining profiled targets. The orienteer then maps each original-program indirect target to its corresponding approximate-program target in the MSSP map table (the reasons for indirect-target mapping are discussed elsewhere [8]).

However, the hardware profiler may not have samples for all the indirect branch targets (either because of sampling error or because of unexecuted paths). If one of these indirect branch targets becomes active after approximate code has been generated for the region, our framework does not currently have a mechanism to detect it. Our preliminary experience suggests that this situation occurs relatively infrequently, but will limit our dynamic coverage in the long run.

We have developed a solution to this problem, but have not implemented it for this work. Our solution involves defining a default stub target for each indirect branch in the approximate program and defining a default mapping entry for the stub in the MSSP map table. If the approximate program tries to execute an unexplored indirect target, our framework will redirect the program to the default stub. The default stub will be profiled with a variation of our profile instruction that samples the intended original-program target and triggers the orienteer if enough samples are collected (at which point the orienteer can either choose to explore or stub the path).

## 3.7 Lessons Learned

In designing the orienteer to address the described examples, we found it useful to adopt several guidelines. The guidelines fulfill a dual purpose of both preventing unexpected program constructs from silently passing through the orienteer and enabling the orienteer to develop a meaningful IR from the program.

The first guideline we adopted was to design our IR with rules that allow the orienteer to detect nontrivial program constructs. For instance, the rule that each basic block in the IR is only a part of a single function helps the orienteer identify eliminated tail calls (Section 3.3) and nested loops (Section 3.5).

Another guideline requires that the orienteer implement detailed error reporting if it does not find a region corresponding with a given hot loop seed. By collecting and analyzing a log of errors we were able to identify false loop seeds and redesign the orienteer to identify and ignore them (as described in Section 3.1).

In general, our guidelines follow the belief that the program orienteer should log any unexpected program characteristics for later analysis. Programs can be written in many different and sometimes convoluted ways, especially when considering compiler-optimized code and libraries implemented in hand-written assembly. Rather than designing the orienteer to address every imaginable program construct, we instead analyze the log generated by the program orienteer in order to investigate and implement solutions for unexpected cases.

# 4 EXPERIMENTAL RESULTS

This section presents an experimental evaluation of our program orienteering framework. The experiments are simulation-based, and a discussion of our methodology follows in Section 4.1. Results are then presented and discussed in Section 4.2.

## 4.1 Methodology

Our simulator implements the MSSP execution paradigm for the Alpha ISA. The simulator models an asymmetric CMP containing many small slave cores and a single large master core. Each core is simulated with a cycle-accurate, event-driven timing model. The simulator also models the on-chip interconnection network latencies of the CMP (although it currently presumes infinite network bandwidth). Table 4.1 details the simulation parameters used in our processor models.

**Table 4.1: Processor Simulation Parameters**

| Parameter | Value |
|---|---|
| Number of Processors | Slave: 8<br>Master: 1 |
| Pipeline | Slave: 2-wide, 6-stage out-of-order<br>Master: 4-wide, 8-stage out-of-order |
| Branch Predictor | Slave: 8 KB, 4-bit history gshare, 32-entry RAS<br>Master: 32 KB, 8-bit history gshare, 32-entry RAS |
| L1 Instruction Cache | Slave: 32 KB, 2-way, 64-byte line size, 8 MSHRs<br>Master: 64 KB, 2-way, 64-byte line size, 8 MSHRs |
| L1 Data Cache | Slave: 64 KB, 16-way, 256-byte line size, 8 MSHRs<br>Master: 64 KB, 2-way, 64-byte line size, 8 MSHRs |
| Unified L2 Cache | Shared: 1 MB, 8-way, 64-byte line size, 512 MSHRs |
| Instruction Queue | Slave: 32 entries; Master: 128 entries |
| Functional Units | Slave: 2 Int, 1 Load/Store, 1 FP<br>Master: 4 Int, 4 Load/Store, 2 FP |
| Interconnect Latency | 5 cycles (one way) |
| Memory Network Latency | 100 cycles (one way) |

The software mechanisms of the program orienteering framework are modeled as a combination of three effects. First, the hardware-collected profile data is scanned for new hot loop seeds every 10 000 cycles, which is an imprecise but likely conservative estimation of a real system. Second, the program orienteering software explores and deploys each new region after a fixed latency of 200 000 cycles, which allots 1600 cycles of work per static instruction explored (on average). Third, when the approximate code regions are generated, we model the L2-cache updates resulting from the orienteer pushing the approximate code to memory. We believe that the result is a conservative latency model for program orienteering behavior.

Our simulator does not currently model the storage mechanisms used by the hardware edge profiler or software profiling instructions. However, the profilers utilize statistical sampling (which reduces throughput requirements on the hardware), and the profile data used a maximum of 11 KB of storage for our runs (4 KB on average). As a result, we do not believe that an accurate model of hardware constraints would dramatically affect our collected results or would cause us to change the design of the program orienteer.

Table 4.2 lists the parameters used for the program orienteer. The hardware profiler sampling rate determines the average number of samples collected per branch visited. We randomly offset the actual sampling rate from the specified value to prevent pathological errors in programs that exhibit periodic behavior exactly in sync with the sampling rate.

**Table 4.2: Program Orienteer Framework Parameters**

| Parameter | Value |
|---|---|
| Hardware Profiler Sampling Rate | 1 sample / 32 branches |
| Hot Loop Seed Scan Interval | 10 000 cycles |
| Hot Loop Seed Threshold | 100 samples |
| Hot Loop Orienteering Latency | 200 000 cycles |
| Stub Adaptation Latency | 50 000 cycles |
| Active Path Bias Threshold | 1% |
| Profile Instruction Sampling Rate | Stub: 1 sample / 1 visit<br>Predecessor: 1 sample / 1 visit |
| Profile Instruction Threshold | Stub: 10 samples<br>Predecessor: 1 000 samples<br>Effective Bias: 1% |

The active bias threshold specifies the minimum relative bias that a profiled path must have so that the orienteer will consider it active. The orienteer will insert stub blocks for insufficiently biased paths.

For this work, the profile instruction parameters control the behavior of the stub adaptation feature described in Section 2.2.2. Recall that the orienteer inserts a pair of profile instructions on each stub path: one into the stub block and one into the predecessor of the stub block. We set the predecessor profile instruction to trigger the removal of the stub path if it collects 10 000 samples at a sampling rate of 1 sample per visit. We set the stub profile instruction to trigger exploration of the stub path if it collects 10 samples at a sampling rate of 1 sample per visit. Therefore, we effectively configure the orienteer to remove stub paths that have less than a 1% bias.

In order to evaluate the program orienteering framework, we simulated 100 million instructions from the SPECint2000 benchmarks. For each benchmark we skipped the first 5 billion instructions in order to avoid startup code. Our framework targets long-running applications, and startup code tends to exhibit drastically different behavior than a program in steady-state.

## 4.2 Results

As mentioned in Section 1.2, any successful dynamic optimization framework must be able to identify the hot regions of a program's execution. In addition, the framework must be able to deploy code that correctly represents these hot regions. We found that our framework meets both criteria.

We first present coverage results for the baseline system described in Section 4.1. Figure 4.1 shows the percentage of dynamic instructions executed inside approximate code regions (dynamic coverage) On average, our baseline framework achieves 94% coverage, which implies that it is successfully finding the hot program regions.

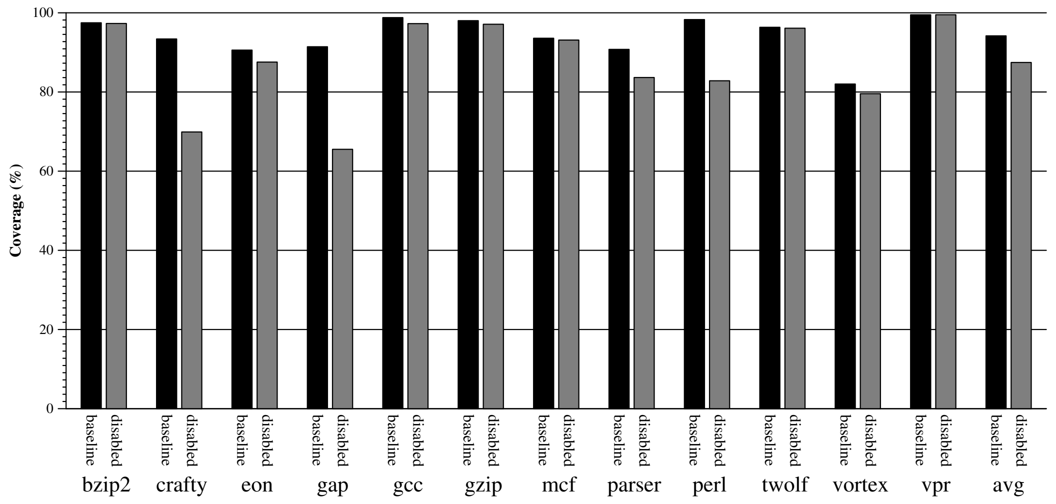

**Figure 4.1: Dynamic coverage of program orienteering framework.** Compares coverage of the baseline framework with the framework with stub adaptation disabled.

Also shown in Figure 4.1, are results for our system with stub adaptation disabled. As discussed in Section 2.2.2, stub adaptation enables the orienteer to explore paths that were deemed cold based on the initial hardware-collected profile. In general the initial hardware-collected profile correctly identifies removable cold paths, but in several benchmarks disabling stub adaptation causes a drop in dynamic coverage. For example, `crafty` suffers a 23% drop in dynamic coverage.

Figure 4.2 compares the dynamic coverage over cycle time in the benchmark `crafty` for our baseline framework versus our framework with stub adaptation disabled. For both configurations, coverage rises sharply after approximately 5 million cycles when the orienteer discovers the hot program region.[1]

When stub adaptation is enabled, coverage continues to rise past 90%. However, coverage eventually tapers off at approximately 70% when stub adaptation is disabled. The coverage problem occurs because the hot program region is a recursive function, which we investigated to be `Search(...)`, containing a large number of executed paths. When the orienteer explores the region, a large number of the paths are stubbed because insufficient

[1]The curve for enabled stub adaptation ends abruptly because the simulation completes in less cycles. The higher dynamic coverage means that the program spends most of its time being executed in the approximate program on the large master processor. The performance improvement comes from two factors: fewer shifts between sequential mode and MSSP mode, and the more aggressive design of the master processor.

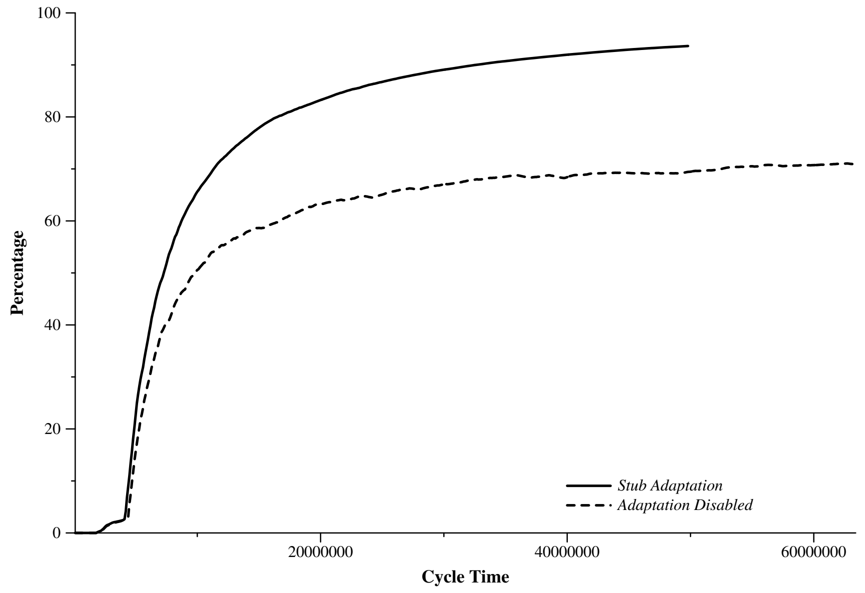

**Figure 4.2: Dynamic coverage timeline comparison for crafty.**

profile samples in the hardware-collected profile make them appear cold. These stub paths are sufficiently executed that the orienteer, with stub adaptation enabled, will explore and include them in the approximate program. However, without stub adaptation the paths are left stubbed, which limits coverage because of frequent exits from MSSP mode (see Appendix A).

As a result, stub adaptation allows us to tune the hardware profiler, using the hot loop seed threshold, such that it quickly collects an initial profile. This approach is sufficient for most hot program regions, but will incorrectly characterize regions with a large number of executed paths such as the recursive function `Search(...)` in `crafty`. Alternatively, we could have set a larger hot loop seed threshold such that the hardware profiler collects more samples in the initial profile. However, setting the threshold higher delays hot loop detection and adversely affects coverage in most benchmarks.

Figure 4.3 shows dynamic coverage results for several settings of the hot loop seed threshold (with stub adaptation disabled in all cases). Increasing the hot loop seed threshold worsens dynamic coverage in the majority of the benchmarks because it is, in general, more beneficial to identify a hot program region early than it is to collect more samples. Increasing the hot loop seed threshold initially improves the dynamic coverage results for `crafty`, but further threshold increases delay hot loop detection enough to adversely affect coverage. As a result, we chose to use a lower hot loop seed threshold and to rely on stub adaptation in order to correct for inaccuracies in the initial profile.

In addition to allowing the orienteer to explore active stub paths, stub adaptation also enables the orienteer to identify removable cold paths. If a path is less than 1% biased, the orienteer will remove the path from the approximate program and remove the associated branch (if it is a conditional direct branch). Figure 4.4 presents the results for removing these cold paths. It shows that, on average, the approximate code generated by our framework covers 96% of the total dynamic branches, and that, by using stub adaptation, the framework was able to speculatively remove highly biased branches accounting for 37% of the total dynamic branches.

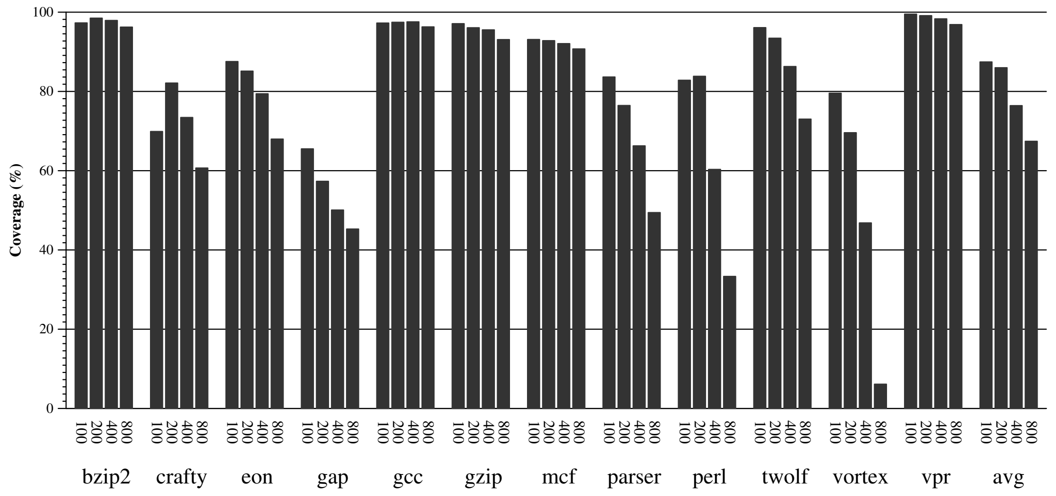

**Figure 4.3: Sensitivity analysis of the hot loop seed threshold.** The numbers below each bar denote the hot loop seed threshold value used.

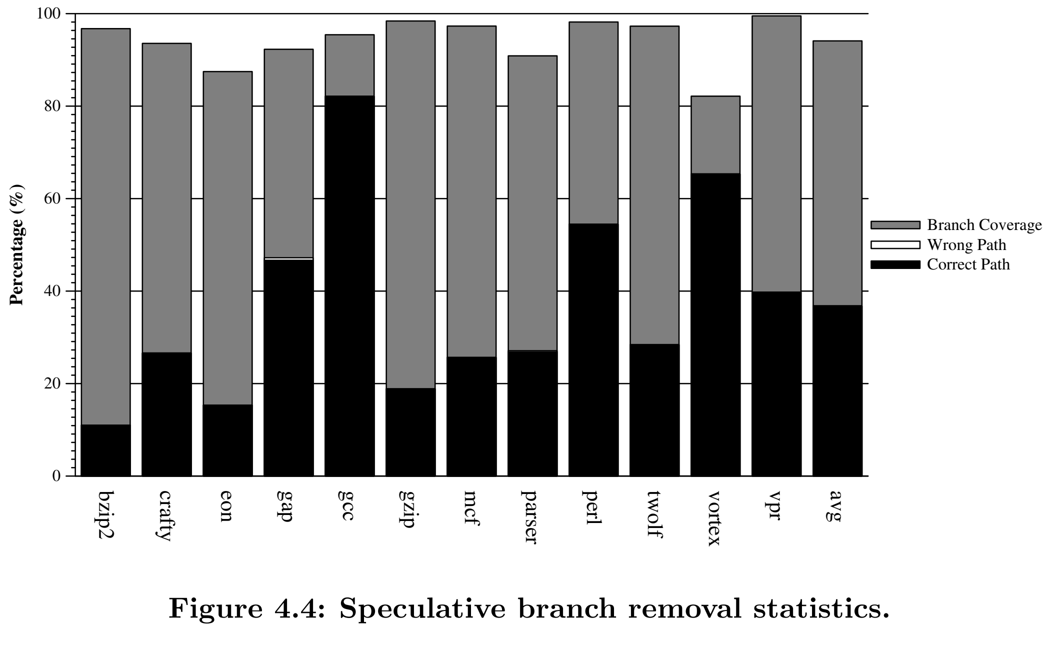

**Figure 4.4: Speculative branch removal statistics.**

The removed branches are still represented in the original program, and thus slave executions will still execute the branch. The slave can execute the removed path, but if it does it indicates an incorrect speculative branch removal. We keep track of the number of times this event occurs and graph it as the *wrong path* percentage in Figure 4.4. The situation is so rare that the wrong path percentage is not even visible on the graph. In general, a wrong path event will cause a slave misspeculation, so we intentionally picked stub profiling thresholds such that wrong path events would be minimized, while still enabling the orienteer to remove highly biased branches.

The system described thus far does has not used the sampling rate field in the profiling instruction. We tested the sensitivity of our framework to using sampled profile instructions in stub predecessor blocks. Figure 4.5 shows the branch removal results for several different sampling rates. Overall the sampling rate has a negligible result on the branch removal accuracy, and reduces the number of profile counter updates significantly. Note that the sampling rates are all powers of 2. This is because the sampling rate is expressed as $1/2^n$ where $n$ is a value encoded in the profile instruction.

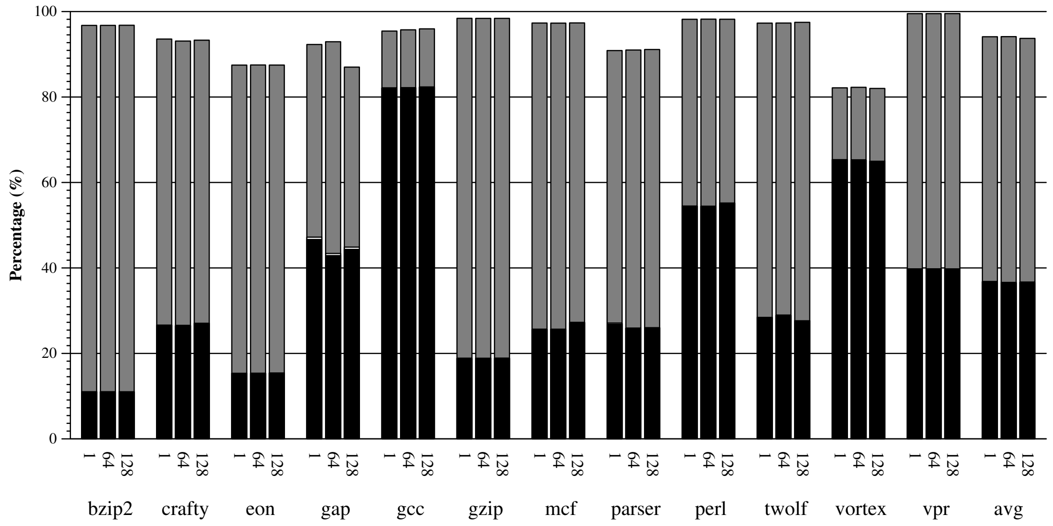

**Figure 4.5: Sensitivity analysis of profile instruction sampling rate.** The numbers below each bar denote the predecessor profile instruction sampling rate used.

However, we believe that stub adaptation implements a naive use of the flexibility provided by the program orienteer and profile instructions. Stub adaptation uses two profile instructions per branch that the orienteer is observing, but the orienteer could optimize the number of profile instructions necessary by placing counters more intelligently. In general, the information provided by a particular profile instruction can be derived from other profile instructions in the region.

In addition, stub adaptation does not use the more flexible features of our profile instruction. The profile instruction encodes a condition field (analogous to a branch condition) that causes the instruction to only increment its associated counter only when the condition has been met. The condition field could be used to compare two register operands and update a counter predicated on the result. Such flexibility in the profile instruction enables techniques such as memory address profiling and value profiling. However, stub adaptation simply sets this field to an unconditional encoding so that each profile instruction increments a counter every time it gets executed (although it can offset this with sampling).

Nevertheless, even our naive use exposes significant optimization potential by identifying highly biased branches that the runtime compiler can remove. After the runtime compiler removes a highly biased branch, it can merge the source and target basic blocks, which exposes additional optimization opportunity to passes such dead-code elimination and register allocation [10]. The result is an approximate program containing optimizations not possible with traditional compiler optimizations alone.

# 5 CONCLUSION

Dynamic FDO frameworks offer significant optimization benefits when coupled to task-level speculation hardware such as a processor implementing the MSSP paradigm. However, the benefits are speculative and have commensurate misspeculation penalties. Therefore, the framework must take great care when speculatively optimizing a program. The framework should be confident that the program behavior presumed by a speculative optimization matches actual program behavior in the common case.

Our proposed mechanism, which we call the program orienteer, does just that. It uses a combination of hardware and software to accurately characterize the hot regions of an executing program. In addition, it represents its characterization in terms of an intermediate representation that can be used by a runtime compiler. We have shown how the program orienteer fits into a dynamic FDO framework, how it uses hardware and software to accomplish its task, and how the information it provides can enable effective use of speculative compiler optimizations.

We are far from completing the implementation of our FDO framework, but the lessons learned so far hint that the program orienteer will play an integral role in our envisioned system. Namely, we expect that the program characterization provided by the orienteer will enable a suite of approximation and optimization transformations to generate a highly optimized approximate program.

In addition, the program orienteer needs further investigation itself. This work primarily focuses on generating an approximate program based on the identification of highly biased branches. Characterizing other aspects of program behavior would likely be beneficial as well. For example, identifying load and store pairs that either almost never alias or almost always alias would aid traditional compiler optimizations such as code scheduling and register allocation of memory variables. Even our approximation of highly biased branches only addresses a subset of a program's control flow, and other classes of branches may offer additional approximation potential. For example, we have found branches that, over time, vary between unbiased and biased, but exhibit biased behavior for long stretches that could be temporarily approximated. The presented program orienteer does not characterize such branches, and thus such approximations currently elude our framework.

Nevertheless, the presented program orienteer conceptually validates our belief that such a mechanism can extract dynamic program behavior in a form that is beneficial to a runtime compiler. We will build on the system developed for this work to address the many remaining issues, and can do so with confidence that our basic ideas are plausible.

# APPENDIX A

# MSSP OVERVIEW

As the name suggests, a Master/Slave Speculative Parallelization (MSSP) execution comprises two program executions: the master and the slave. The master execution speculatively runs ahead and is responsible for the performance of the overall execution. The slave execution, which is orchestrated by the master, is responsible for the correctness of the whole execution and verifies the master's speculation.

The master execution is performed by a single master processor. The master executes a version of the program (the *approximate program*) where predictable computations are removed. The predictable behaviors are removed via approximation transformations; unlike traditional compiler transformations which must preserve all potential program behaviors, approximation transformations are allowed to violate correctness. By leveraging an accurate program characterization, we can apply these transformations such that the common case performance is improved and correctness violations are minimized. In addition, approximation transformations typically create new opportunities for traditional optimizations, so our envisioned system follows approximation with a suite of traditional compiler optimizations.

The result of approximation is a program that runs substantially faster than the original program but with no guarantees of correctness. To ensure correctness, we complement the master's execution of the approximate program with an execution of the original program to verify the predictions. The key challenge is to perform the verification in a manner that it does not become the bottleneck. MSSP parallelizes the execution of the original program onto many slave processors to achieve the throughput necessary to keep up with the master.

The execution of the original program is split into segments, called *tasks.* To enable these tasks to execute independently and in parallel, the master execution is used to predict the sequence of tasks (i.e., the starting program counter (PC) of each task) and the live-in values to these tasks. In fact, generating these predictions is the master's only responsibility. The register writes and stores performed by the master are held in a special *checkpoint buffer*, and the predictions are generated by logically taking a checkpoint of the master's state at the point corresponding to the beginning of the task. When the master's state is no longer necessary it is discarded; the master never directly modifies the architected state.

The approximate program specifies the beginning of a task with a `fork` instruction that encodes the starting address for a slave execution. When the master executes the `fork` instruction it spawns a slave task, which begins execution at the specified address. The master then continues executing along the fallthrough path of the `fork` instruction.

In practice, the slave's starting address resides in the approximate program and specifies the beginning of a `verify` block. The `verify` block is terminated by a `verify` instruction

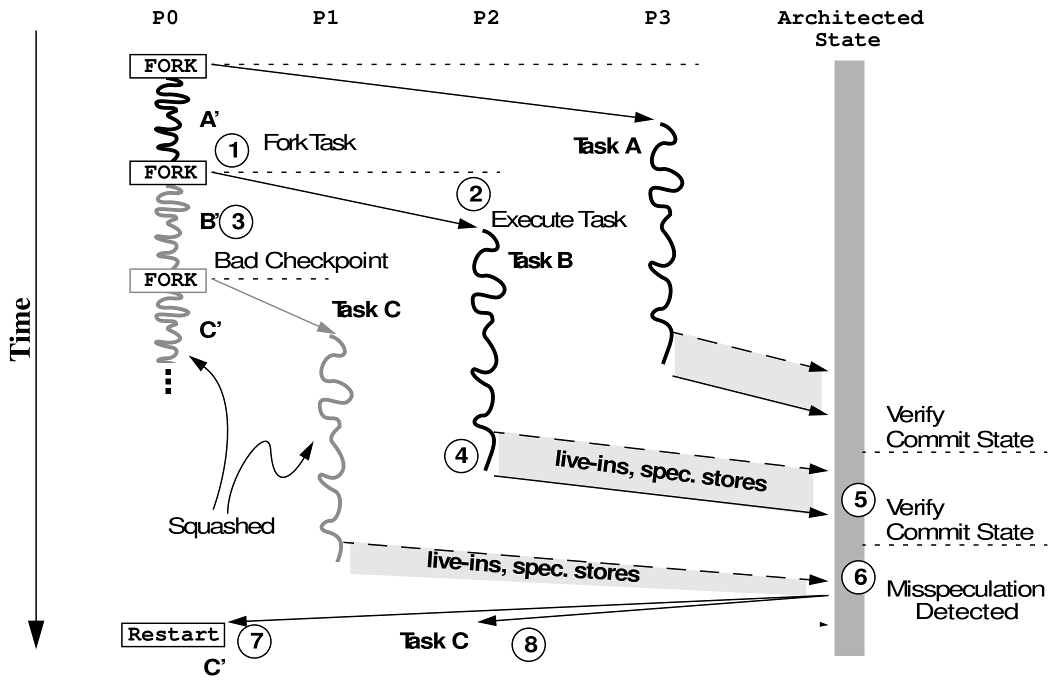

**Figure A.1: Master processor distributes checkpoints to slaves.** The master—executing the approximate program on processor P0—assigns tasks to slave processors, providing them with predicted live-in values in the form of checkpoints. The live-in values are verified when the previous task retires. Misspeculation, due to incorrect checkpoint, causes the master to be restarted with the correct architected state.

Figure A.1 illustrates an MSSP execution with four processors. One processor (P0) is allocated to be the master processor, and the remaining processors (P1, P2, and P3) are slaves that begin the example idle. The master executes the approximate program and, at a `fork`, spawns a new task on an idle slave processor and provides it access to the buffered checkpoint values. At annotation ① in the figure, the master processor spawns Task B onto processor P2. P2 begins executing the task after some latency due to the interprocessor communication ②. P0 continues executing the approximate program segment that corresponds to Task B, which we refer to as Task B′ ③.

As the slave Task B executes, it will read values that it did not write (live-in values) and perform writes of its own (live-out values). If a corresponding checkpoint value is available, it is used for the live-in value; otherwise, the value is read from the currently visible (architected) state. As the slave tasks are speculative—they may be using predicted live-in values—their live-out values cannot be immediately committed: we have to ensure the live-in values are correct before committing the live-outs. To this end, we record the task's live-in and live-out values. When all previous tasks have completed and updated the architected state, the live-ins can be compared with the architected state. To avoid interprocessor communication in the verification critical path, our implementation performs this comparison at the (banked) shared level of the cache hierarchy. Thus, when the task is complete ④, P2 sends its live-in and live-out values to the shared cache. If the recorded live-in values exactly correspond to the architected state, then the task has been verified and can be committed, and the architected state can be updated ⑤ with the task's live-out values.

If one of the recorded live-in values differs from the corresponding value in the architect state (e.g., because the master wrote an incorrect value ③), this mismatch will be detected during verification. On detection of the misspeculation ⑥, the master is squashed, as are all other in-flight tasks. At this time, the master is restarted at $C'$ ⑦ using the current architected state. In parallel, nonspeculative execution of the corresponding task in the original program (Task C) begins ⑧.

In this work, the approximate program is generated by a dynamic FDO framework. As such, when the program starts no approximate program will exist and execution will begin with a sequential execution of the original program. When the framework generates an approximate program region, it will also update a MSSP map table that maps original program addresses to entry points in the approximate program. When the sequential execution encounters an original program address that maps into the approximate program, execution will transition into MSSP execution mode. Entry points are typically placed at the beginning of loops and functions.

The program's execution will stay in MSSP execution mode until either a misspeculation causes a restart[1] or the master executes a `master_end` instruction. The `master_end` instruction terminates the master execution without causing a misspeculation. After all the remaining speculative slave tasks are verified, execution resumes in sequential execution mode. Our framework places `master_end` instructions on paths that are suspected to be cold, so that the paths can be stubbed out of the approximate program. However, if the stub path is needed, a misspeculation will not occur, and, instead, the system will smoothly transition into sequential execution mode.

In conclusion, MSSP can bring a significant amount of silicon real estate to bear on the execution of a sequential program, even in technologies that are communication limited. The individual cores are sized for efficient communication and *all of the latency intolerant communication occurs within the processor cores.* The intercore communications only contribute to the latency of verifying the predictions made by the execution of the approximate program; if the approximate program is constructed such that these predictions are accurate, this latency can be effectively tolerated.

[1]Actually execution will stay in MSSP mode even after a restart if a map table entry for transitioning from the original program to the approximate program exists at the restart address.

# REFERENCES

[1] A. Chernoff, M. Herdeg, R. Hookway, C. Reeve, N. Rubin, T. Tye, S. B. Yadavalli, and J. Yates, "FX!32: A profile-directed binary translator," *IEEE Micro*, vol. 18, no. 2, pp. 56–64, Mar 1998.

[2] R. Cohn, D. Goodwin, and P. G. Lowney, "Optimizing Alpha executables on Windows NT with Spike," *Digital Technical Journal*, vol. 9, no. 4, pp. 3–20, 1997.

[3] J. C. Dehnert, B. K. Grant, J. P. Banning, R. Johnson, T. Kistler, A. Klaiber, and J. Mattson, "The Transmeta Code Morphing$^{TM}$software: Using speculation, recovery and adaptive retranslation to address real-life challenges," in *1st IEEE/ACM Symp. Code Generation and Optimization*, San Francisco, CA, Mar 2003, pp. 15–24.

[4] K. Ebcioglu and E. R. Altman, "DAISY: Dynamic compilation for 100% architectural compatibility," in *Proceedings of the 24th Annual International Symposium on Computer Architecture*, June 1997, pp. 26–37.

[5] V. Bala, E. Duesterwald, and S. Banerjia, "Dynamo: A transparent dynamic optimization system," in *Proceedings of the SIGPLAN 2000 Conference on Programming Language Design and Implementation*, June 2000, pp. 1–12.

[6] L. Baraz, T. Devor, O. Etzion, S. Goldenberg, A. Skaletsky, Y. Wang, and Y. Zemach, "IA-32 execution layer: A two-phase dynamic translator designed to support IA-32 applications on Itanium©-based systems," in *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, Dec 2003, pp. 191–202.

[7] M. Arnold, S. J. Fink, D. Grove, M. Hind, and P. F. Sweeney, "Adaptive optimization in the Jalapeño JVM," in *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages and Application (OOPSLA)*, Oct. 2000, pp. 47–65.

[8] C. Zilles, "Master/slave speculative parallelization and approximate code," Ph.D. dissertation, Computer Sciences Department, University of Wisconsin–Madison, Aug. 2002.

[9] C. Zilles and G. Sohi, "Master/slave speculative parallelization," in *Proceedings of the 35th Annual IEEE/ACM International Symposium on Microarchitecture*, Nov. 2002, pp. 85–96.

[10] S. S. Muchnick, *Advanced Compiler Design and Implementation.* San Francisco, CA: Morgan Kaufman, 1997.

[11] J. Anderson, L. Berc, J. Dean, S. Ghemawat, M. Henzinger, S.-T. Leung, R. Sites, M. T. Vandevoorde, C. A. Waldspurger, and W. E. Weihl, “Continuous profiling: Where have all the cycles gone?” in *Proc. 16$^{th}$ Symposium on Operating System Principles*, Oct. 1997, pp. 1–14.

[12] T. H. Heil and J. E. Smith, “Relational profiling: Enabling thread level parallelism in virtual machines,” in *Proceedings of the 33$^{rd}$ Annual IEEE/ACM International Symposium on Microarchitecture*, Dec. 2000, pp. 281–290.