Appears in the proceedings of the 35th International Symposium on Microarchitecture (Micro-35), November 20-22, 2002

Master/Slave Speculative Parallelization

Craig Zilles
Department of Computer Science
University of Illinois at Urbana-Champaign
zilles@cs.uiuc.edu

Gurindar Sohi
Computer Sciences Department
University of Wisconsin at Madison
sohi@cs.wisc.edu

Abstract

Master/Slave Speculative Parallelization (MSSP) is an execution paradigm for improving the execution rate of sequential programs by parallelizing them speculatively for execution on a multiprocessor. In MSSP, one processor—the master—executes an approximate version of the program to compute selected values that the full program's execution is expected to compute. The master's results are checked by slave processors that execute the original program. This validation is parallelized by cutting the program's execution into tasks. Each slave uses its predicted inputs (as computed by the master) to validate the input predictions of the next task, inductively validating the entire execution.

The performance of MSSP is largely determined by the execution rate of the approximate program. Since approximate code has no correctness requirements (in essence it is a software value predictor), it can be optimized more effectively than traditionally generated code. It is free to sacrifice correctness in the uncommon case to maximize performance in the common case.

A simulation-based evaluation of an initial MSSP implementation achieves speedups of up to 1.7 (harmonic mean 1.25) on the SPEC2000 integer benchmarks. Performance is currently limited by the effectiveness with which our current automated infrastructure approximates programs, which can likely be improved significantly.

1 Introduction

Most microprocessor vendors are shipping or have announced products that exploit explicit thread-level parallelism at the chip level either through chip multiprocessing (CMP) or simultaneous multithreading (SMT). These architectures are compelling because they enable efficient utilization of large transistor budgets—even in the presence of increasing wire delay—for multi-threaded or multi-programmed workloads. Although we expect the availability of these processors to encourage some programmers to explicitly parallelize their programs, anecdotal evidence that many software vendors ship un-optimized binaries suggests that many programmers cannot justify (either to themselves or their employers) the additional effort of correctly parallelizing their code. As a result there will remain an opportunity for "transparent" parallelization.

Transparent parallelization can be achieved by analyzing the program's dependences, partitioning the program into independent subsets (*tasks*), and inserting the necessary synchronization. For non-numeric programs, a complete dependence analysis is prohibitively difficult. As a result, parallelization of these programs is facilitated by speculating in the presence of ambiguous dependences and providing hardware support for the detection of and recovery from actions that violate the ordering dictated by a sequential execution. Many proposals for speculatively parallelizing programs have been published [1, 6, 8, 12, 20, 23].

Speculation allows the execution to ignore potential dependences that do not occur in practice, but it does little to alleviate true dependences. True inter-task data dependences can sequentialize task execution, negating much of the performance potential of speculative parallelism. In fact, because the inter-task dependences generally incur the latency of inter-processor communication, performance can be even worse than that of a uniprocessor execution. Thus, one of the main challenges remaining in speculative parallelization is handling true dependences.

In this paper, we present an execution paradigm for speculative parallelization that breaks inter-task true dependences by predicting the communicated values. This approach, in itself, is not novel [1, 15, 16, 21]; what is novel is the manner in which the predictions are made. Rather than use a history-based hardware widget, our approach uses a piece of code that, when executed, predicts the values communicated between tasks.

We call this code the *distilled program* because it is created by "distilling" the original program down to a reduced computation that will likely compute the necessary values. Since results computed by the distilled program are only used as predictions, there are no correctness constraints on the distilled program; all predictions will be validated before they are allowed to alter architected state. This freedom from correctness constraints bestows unlimited flexibility to the construction of the distilled program, allowing us to guide optimizations solely based on common case program behavior rather than what could be proven correct statically. As a result, the distilled program can significantly outperform, while closely paralleling in function, the original program, but it provides no correctness guarantees.

In our execution paradigm there are two distinct processor[1] roles. One processor, the *master*, is assigned to execute the distilled program. This execution orchestrates a parallel execution of the original program on the remaining *slave* processors. At defined places in the distilled program's execution, the master "spawns" a task onto an idle slave processor, providing it with a starting program counter (PC) and predictions for the live-in values. The slave processors execute the un-modified original program, and only they update architected state. State updates are speculatively buffered until the task has been verified. Values produced by the master are discarded when no longer needed as live-in predictions. The relationship between master and slave is reminiscent of that between a speculative core and a DIVA checker [2], except with the purpose of detecting errors in code transformation rather than processor design.

We call this execution paradigm Master/Slave Speculative Parallelization (MSSP). We first provide an overview of the opportunity for distilling programs and the MSSP paradigm in Section 2. In Section 3, we detail the automatic construction of distilled programs using profile information. In Section 4, we describe the mechanisms required by MSSP and provide a detailed example of its execution. In Section 5, we present results from an initial simulated implementation of MSSP. This paradigm has been heavily influenced by prior work, which is discussed throughout the paper.

2 Overview

In this section, we introduce the concept of code approximation used to create distilled programs and provide a high-level overview and analysis of the MSSP paradigm.

2.1 Code Approximation

The MSSP paradigm predicts all of a task's live-in values (*i.e.*, those values that are used before they are defined). Unlike most value predictors, the one used in MSSP is a piece of software, called the distilled program, that produces predictions when executed. Like any predictor, we desire that it have a high prediction accuracy, but mistakes can be tolerated. The distilled program is constructed by *approximating* the program being executed, which we will refer to as the original program.

Approximation exploits the fact that most programs are more general than is required by any particular execution. As a result, much of a program's functionality is not needed in the common case. This observation has been exploited previously in feedback-directed program transformations that optimize the common case at the expense of the uncommon case. Approximation is an extreme form of such transformations that further optimizes the common case by sacrificing correctness in the uncommon case.

1. For simplicity of exposition we use the word "processor", but the ideas are equally applicable to multithreaded processors.

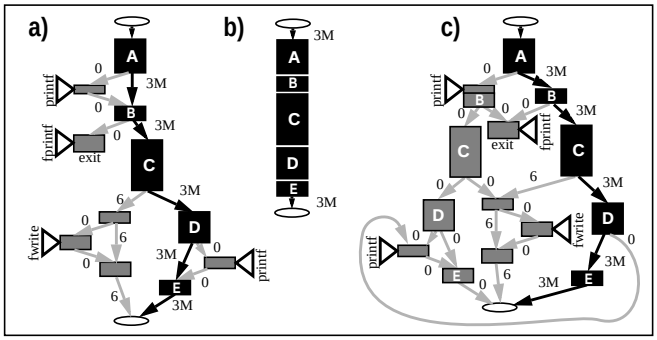

Figure 1. The control flow graph (CFG) for the function `spec_getc` from the benchmark `bzip2`: a) initially, b) as transformed by approximation, c) as transformed by superblock formation.

Approximation is accomplished by removing behaviors from the program that appear not to be frequently exercised. This process is most easily illustrated through an example: Figure 1a shows a control flow graph (CFG) of a small function annotated with an edge profile. In approximating the function, non-dominant control flow edges are removed. As this function has a single dominant path, after approximation the function is a single block (Figure 1b). In general, this will not be the case; unbiased branches along important paths will remain. Approximation transformations are not limited to control-flow manipulations; others are described in Section 3.3.

Approximation transformations are like speculative compiler transformations but without the checks that ensure the expected behavior is present. The dominant path in this example can also be exploited by superblock formation [13] resulting in the CFG shown in Figure 1c. While superblock formation removes side entrances from the trace (by replicating code), side exits must be left intact, as they are required to detect and handle the correct execution of non-dominant paths. In the approximate code, these checks (the branches) are removed, resulting in an incorrect computation when a non-dominant path execution is required.

These approximation transformations create new opportunities to apply traditional (safe) optimizations. By eliminating dead code (mostly the branch predicate computations) and avoiding stack allocation, saves and restores (the approximate function has lower register pressure and is now a leaf function), the dominant path through the function in Figure 1 is reduced from 51 to 12 instructions. This small static size facilitates inlining, which can create additional opportunities for optimization.

Approximation has many similarities to the instruction removal performed in Slipstream [22]; unlike Slipstream, we create a distinct executable to enable traditional optimizations to be applied as well. A process akin to approximation has been used to create predictors for branches and memory addresses [7, 18, 25]. The particular form of approximate code used in the MSSP paradigm—what we call a distilled

program—also encodes the information necessary to manage the parallel execution of the slave processors. We discuss the construction of distilled programs in Section 3, after an overview of the MSSP paradigm.

2.2 MSSP Overview

We now describe the MSSP paradigm at a high level; a description of the required mechanisms and a more detailed example are presented in Section 4. Like other speculative parallelization paradigms, MSSP executes coarse-grain units of work, called tasks, generally consisting of hundreds of instructions. Task boundaries are selected during the construction of the distilled program and encoded within it.

An MSSP execution consists of three separate components, each with a different purpose: 1) the master processor executes the distilled program to compute predicted *checkpoints* of state at task boundaries, 2) using these checkpoints, slave processors execute from one task boundary to the next in the original program constructing a *summary* of the task, and 3) a reuse-like [19] mechanism verifies the correctness of the task summaries and commits their results in order. These roles are demonstrated by the example four-processor MSSP execution shown in Figure 2a.

One processor (*P0*) is assigned to be the master processor and executes the distilled program. As the master executes, it performs register and memory writes, but these writes are not committed to architected state. Instead, when a write is "retired" by the master, it is held in a special buffer and tagged with the current task number. The writes for a given task form the change of state (the difference or *diff*) effected by the task. A predicted future checkpoint of state can be constructed by overlaying the differences associated with in-flight tasks (in order from oldest to youngest) on top of the architected register and memory state, as shown in Figure 2b. Checkpoints can be constructed piece-wise and on demand, so the amount of storage required is a function of task working set size and not the size of the program's memory image.

The remaining processors (*P1*, *P2*, and *P3*) are slaves that are initially idle. As directed by the distilled program, the master periodically spawns tasks for the slave processors to execute. To spawn, the master increments its task number, selects an idle slave processor, and provides the slave three inputs: 1) the master's current task number, 2) a starting program counter (PC), and 3) access to the predicted checkpoint state. At marker ❶ in Figure 2a, the master processor spawns *Task A* onto processor *P3*. In parallel with the spawning process, *P0* continues execution of the distilled program, entering the distilled program segment that corresponds to *Task A*, which we refer to as *A´*. *P3* begins execution (❷) some time later as the spawn involves inter-processor communication.

As the slave executes *Task A*, it retrieves any necessary live-in values (*i.e.*, registers and memory locations that are read before they are written by the task) from the checkpoint. Since the values retrieved from the checkpoint are predictions, writes by the slave are not directly committed to archi-

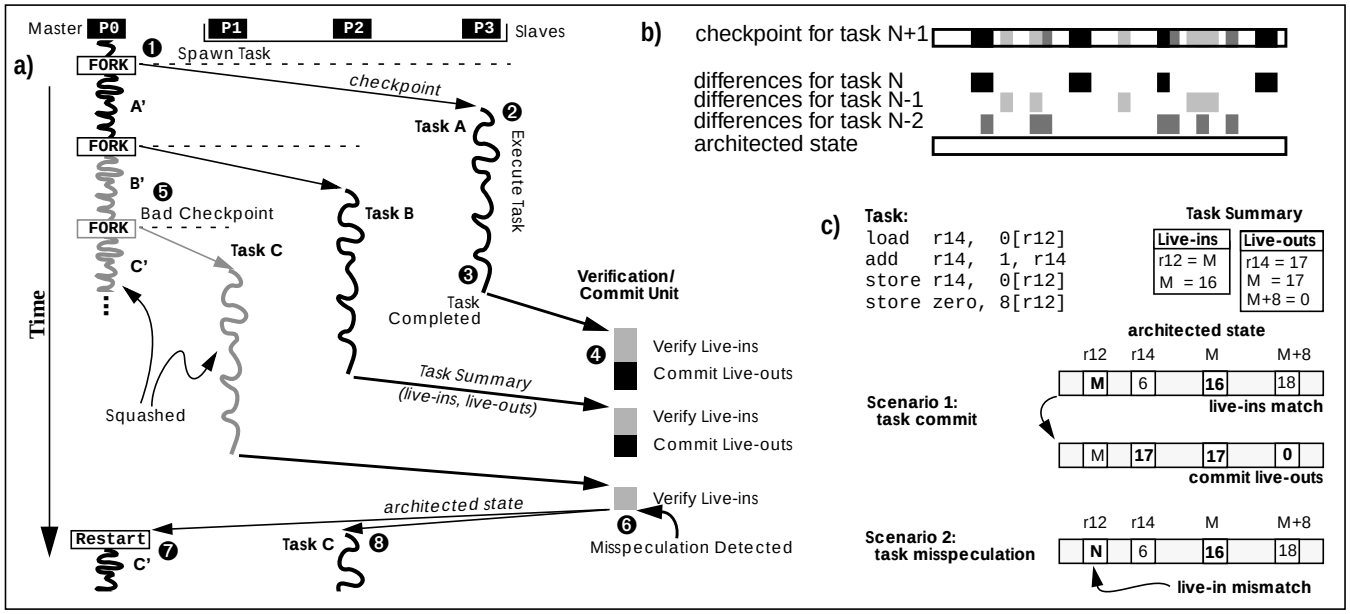

Figure 2. MSSP execution example: a) The master, executing the distilled program on processor P0, forks tasks, providing them live-in values in the form of checkpoints. The slave processors execute the original program to construct task summaries, which are verified before they are committed. Misspeculations, resulting from incorrectly computed checkpoints, cause the master to be restarted with the architected state. b) Writes by the master are buffered as checkpoint differences, which together with architected state can be used to construct predicted checkpoints of future state. c) Task execution is summarized by its set of live-in and live-out values; summaries are verified by comparing the live-ins to architected state. If the live-ins match (Scenario 1), the architected state is updated by committing the live-outs. Otherwise (Scenario 2), a task misspeculation is signalled.

tected state. Instead, the live-ins and live-outs (the last write to each location written by the task) are collected (both name and value) and buffered as a task summary.

When the slave completes its task (❸), the task summary is sent to the verification/commit unit. This unit tries to commit the summaries in program order, as indicated by the associated task number. The verification/commit process is much like the reuse test [19] or memoization. As a precondition for committing the task, all of the live-in values used in the task's execution must match the value held in architected state. If this condition holds, the live-out values can be committed to architected state (❹). When a task is committed the corresponding checkpoint differences and task summary can be deallocated.

If one or more of the live-in values do not match with architected state (because a checkpoint was computed incorrectly (❺)), a task misspeculation is signalled. On detection of the misspeculation (❻), the master is squashed, as are all other in-flight tasks. At this time, the master is restarted at *Task C´* (❼) using the current architected state committed by *Task B*. In parallel, execution of the corresponding task in the original program (*Task C*) begins (❽). The recovery process is expensive, but it is infrequent when using well-constructed distilled programs.

The verification/commit process is shown in Figure 2c. Note that the checkpoints themselves are never verified directly. Only the task live-ins (*i.e.*, those values that could corrupt architected state) are checked. Distilled program construction exploits this fact to avoid computing any value that has a low likelihood of being a task live-in.

2.3 Analysis of MSSP Execution

Several characteristics distinguish MSSP from other speculative parallelization paradigms. In programs with significant amounts of true inter-task dependences (*e.g.*, most non-numeric programs), the critical path of a traditional speculative parallel execution follows sequentially-dependent communications to forward values (Figure 3a). Long inter-processor communication latencies reduce the degree to which task executions are overlapped.

In contrast, in an MSSP execution, values are not communicated between slave tasks. The production of values to satisfy inter-task dependences is decoupled from the production of values to update architected state. All inter-task true dependences are predicted from a central source, the master. As a result, inter-processor communication for different tasks can be performed in parallel, and this communication appears on the critical path only on a value misprediction. Trace processors [17] similarly proposed using value prediction to tolerate communication latency between processing elements.

If mispredictions can be made rare, then the critical path will be through the slower of two paths: the master's execution and the verification/commit unit (Figure 3b). Neither

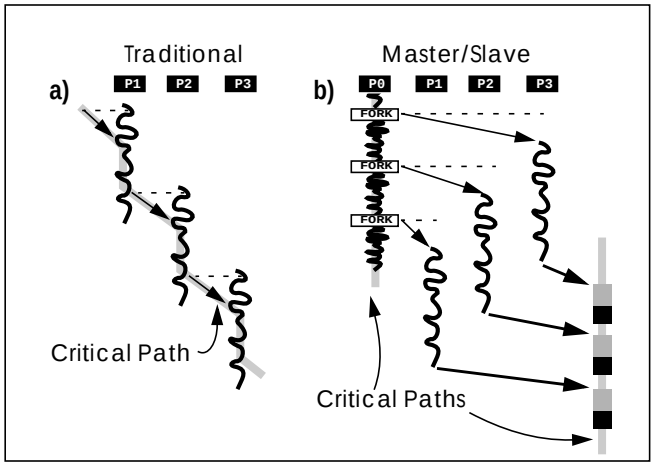

Figure 3. In a traditional speculative multithreading execution model, the critical path of the execution typical goes through values that are forwarded from one task to the next (a). MSSP has two parallel potential critical paths, the execution of the distilled program and verification/commitment (b).

path sequentializes inter-processor communication, providing MSSP with a degree of latency tolerance. Increasing communication latency only increases the occupancy of slave processors and verification latency (not throughput), which is only exposed on task misspeculations.

To summarize, the first requirement of an effective MSSP execution is that the distilled program accurately predict task live-in values. If that is achieved, the master will determine the execution rate of the whole execution, so the distilled program should execute much faster than the original program. Finally, so as not to become the bottleneck, the throughput of the verification/commit unit must exceed the rate tasks are spawned by the master. These first two issues are a function of the construction of distilled program (discussed in the next section). The verification/commit mechanism is described in Section 4.

3 Distilled Programs

While in the previous section we indicate that the distilled program should have a high prediction accuracy and be fast, there are absolutely no requirements on the distilled program necessary to ensure a correct execution. It is only a predictor, and, if it is behaving improperly, the execution reverts to a traditional uniprocessor execution. Nevertheless, MSSP's performance depends on the distilled program performing the following two tasks correctly most of the time: 1) specifying a division of the program's execution into tasks by denoting task boundaries, and 2) accurately computing the set of live-in values at these task boundaries.

In this section, we describe a technique to construct distilled programs. The resulting distilled program will be located in a separate region of memory than the original program. Distillation consists of the seven steps listed in

Figure 4. The first two steps (collect profile information and build IR) and the last step (layout and generate code) are largely unchanged from traditional compiler techniques, so our description will focus on the other steps. Since approximation transformations are likely more sensitive to the accuracy of profile information than traditional feedback-directed program transformations, it might prove necessary to perform distillation at run-time (when the most representative profile information is available), like a dynamic optimizer (*e.g.*, Dynamo [3]).

We illustrate the process of constructing a distilled program with a hot loop nest from `gcc`, shown in Figure 5. The variable `sometimes_max` is a loop-carried dependence, but, as shown in the control flow graph in Figure 6a, the inner loop is only executed on 0.1 percent of outer loop iterations.

3.1 Task Selection

Our sensitivity analysis results (in [24]) indicate our current automatic distiller prototype is largely insensitive to the exact task boundaries selected (performance varies less than 5% for most benchmarks) provided the tasks are large enough (greater than 100 instructions on average) to amortize the communications overheads. As a result, we only highlight a few important issues here; a complete description of our task selector can be found in [24].

Tasks are selected by identifying instructions in the original program to be the beginning of new tasks. A fork instruction is inserted into the distilled program at the task

1. collect profile information
2. build internal representation (IR)
3. select task boundaries
4. perform liveness analysis
5. apply approximation transformations
6. apply traditional optimizations
7. layout and generate code

Figure 4. Logical steps to build a distilled program.

```
for (i = 0; i < regset_size; i++) {
  register REGSET_ELT_TYPE diff = live[i] & ~maxlive[i];   } A
  if (diff) {
    register int regno;                                    } B
    maxlive[i] |= diff;
C{  for (regno = 0; diff && regno < REGSET_ELT_BITS; regno++) {
      if (diff & ((REGSET_ELT_TYPE) 1 << regno)) {
        regs_sometimes_live[sometimes_max].offset = i;
    D{  regs_sometimes_live[sometimes_max].bit = regno;
        diff &= ~ ((REGSET_ELT_TYPE) 1 << regno);
        sometimes_max++;
      }
  E }
  }
} F
```

Figure 5. Example code fragment from SPEC 2000 benchmark `gcc`. Loop nest extracted from function `propagate_block` (lines 1660-1678 of `flow.c`).

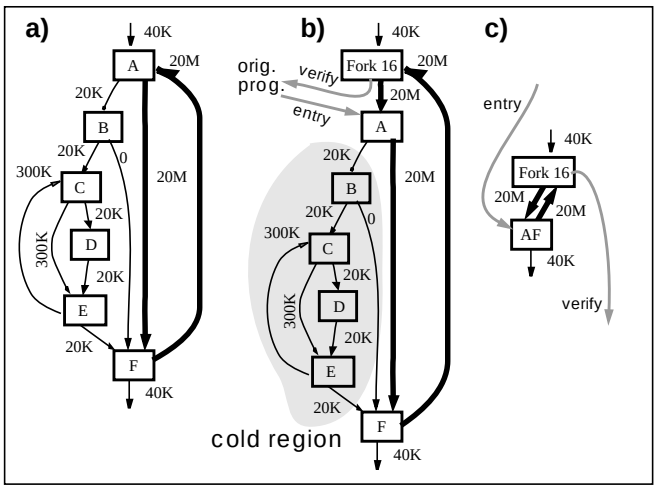

Figure 6. Example task selection and application of approximation transformations. The control flow graph for this code example is shown for: (a) the original code, (b) the code after it has been transformed to include a task boundary, and (c) after the cold code region has been removed.

boundary so it can signal the beginning of a new task. The original program is annotated[2] with these task boundaries for use by slave processors to identify the end of a task. For simplicity of implementation, tasks unconditionally end immediately before annotated instructions, except in one circumstance.

The problem with unconditional task boundaries is demonstrated by our example. We would like to use the frequently executed outer loop for task selection, but with unconditional task boundaries we are left with the sub-optimal choice between: small, single iteration tasks (13 instructions on average) and very large tasks that encapsulate the whole loop (6,500 instructions on average). We resolve this relatively frequent case by creating conditional task boundaries; a technique we call *task end suppression* enables tasks to contain multiple (but not all) iterations of a loop by identifying a task boundary that should be ignored a specified number of times. In our example, the task boundary is inserted in the loop header of the outer loop, and tasks are specified to contain 16 iterations.

During task selection, the CFG is transformed to add logical edges to and from the original program (shown in Figure 6b). These edges maintain our ability to transition between the distilled and original program at task boundaries. In order to commence execution of the distilled program (either initially or after a task misspeculation), an *entry* CFG edge is added. To support spawning the tasks in the original program, a *verify* CFG edge is added. Both edges link to the corresponding location in the original program.

2. The ISA need not be modified to support these annotations by locating them in a separate region of memory that is merged with instructions on an I-cache miss [20].

3.2 Liveness Analysis

After task selection and before optimization, liveness analysis is performed to determine the set of live values at task boundaries. These values should be computed by the distilled program (to act as live-in value predictions) even if all their uses in the distilled program are removed. As this liveness analysis is only used in constructing the distilled program, it need not be sound; for example, we can ignore infrequent paths. The verify arc can be treated as using all of the live values at a task boundary, allowing traditional formulations of data-flow algorithms to be used.

3.3 Approximation Transformations

Approximation transformations are the key to making the distilled program faster than the original program. In Section 2.1, we demonstrated transformations to remove biased branches. Our example code fragment benefits from the elimination of the cold region containing the inner loop. By eliminating these paths, we are left with a greatly simplified CFG (shown in Figure 6c).

In addition to control-flow transformations, value-based (*e.g.*, constant substitution for invariant values) and dependence-based (*e.g.*, ignore may-alias for load-store pair that rarely alias) approximations can be applied. For most any speculative optimization (*e.g.*, [5, 10, 11]), an approximation analog can be formulated.

Furthermore, the redundant nature of MSSP execution enables the optimization of writes with "distant" first uses. If the first read of a write's value is sufficiently far (in dynamic instructions) after the write, the write need not be performed by the distilled program. By the time the read is executed, the (original program) copy of the write executed by a slave processor will have been committed and the value can be retrieved from architected state. Our implementation of *distant-use store elimination* finds many opportunities; for the SPEC2000 integer benchmarks, about 35 percent of dynamic stores have first-use distances exceeding 10,000 instructions.

3.4 Enabled Traditional Optimizations

Although they may remove some instructions from important paths, the main contribution made by the approximation transformations is to create new opportunities to apply traditional (safe) optimizations. By removing cold paths and other features that prevent optimization, even optimized binaries benefit from a re-application of techniques like dead code elimination, partial redundancy elimination, and register allocation.

In our running example, after approximation, half of the code in the loop is dead and can be trivially removed (Figure 7a). Not all optimizations are so straightforward. Instruction F4 is loop invariant and can potentially be hoisted to a loop pre-header block (block *P* in Figure 7b), but this code motion crosses the entry edge in the CFG. To compensate for the code motion, a copy of the invariant load is introduced as compensation code in a block on the entry edge. This entry block translates the state of the original program to the state expected by the distilled program. Some optimizations (*e.g.*, partial dead code elimination) introduce compensation code onto the verify edge.

As a final optimization, the loop in our example is unrolled (by a factor of 16) in the distilled program. Since the induction variables are dead on exit from the loop this unrolling can be accomplished by merely scaling the immediates of instructions F1, F2, and F3. The final code is shown in Figure 7c. Distillation has reduced this task from executing over 200 dynamic instructions on average in the original program to only six in the distilled program.

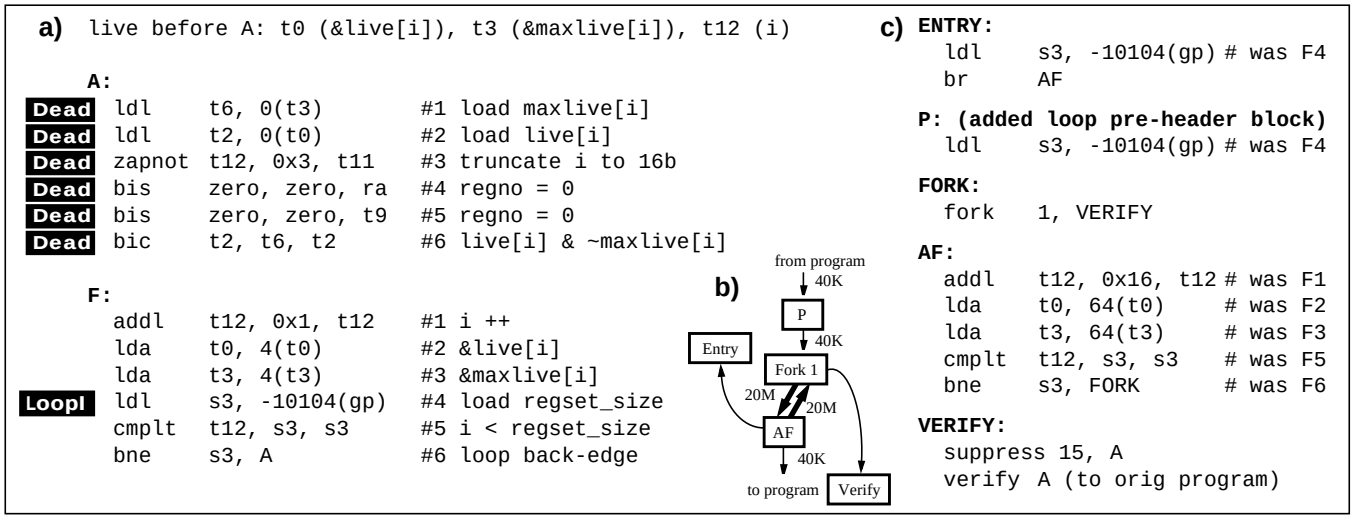

Figure 7. Distilled program fragment after application of approximation transformations (a). After the removal of the branch at the end of block A, instructions A1-A6 are dead. Instruction F4 is loop invariant. To hoist F4, a loop pre-header block *P* is created (b), and a copy of the instruction is added to the entry block. The final optimized code is shown in (c).

3.5 Forking, Verifying, Suppressing, and Mapping

We now briefly touch on our implementation of the MSSP-specific operations in the distilled program, which can be used to decipher Figure 7c. `Fork` instructions are a branch-style instruction that specify the PC of a verify block where a slave processor should begin execution and the denominator of the fraction of times the fork should be performed (*i.e.*, 1 → always, 16 → every 16th iteration). Not spawning at every fork instruction is necessary in some instances when task end suppression is used.

This verify block contains any compensation code necessary and is terminated by a `verify` instruction. The `verify` instruction encodes the original program PC where the task should begin and communicates to the implementation that task summarization (*i.e.*, collection of live-in and live-out values) should begin. If task end suppression is being used, a `suppress` instruction is included in the verify block that encodes the task boundary to suppress and the number of times it should be ignored.

Since the distilled program is distinct from the original program, a mechanism is required to map program counters (PCs) between the two programs. This requirement is shared by dynamic translators and optimizers (*e.g.*, [3]), and our solution is similar. We construct a hash table of PC-pairs that allows searching for "entries" and indirect branch targets in the distilled program using an original program PC. Mapping in the reverse direction is simpler, because the situations that require translation (`verify` instructions and the link part of jump-and-link instructions) can be translated statically (at distillation) and the translated PCs can be encoded into the distilled program as immediates.

4 Implementing the MSSP Paradigm

In this section, we first present an example of MSSP execution. Generalizing from this example, we describe the mechanisms that are essential to the MSSP paradigm in Section 4.2. In Section 4.3, we explain that MSSP's functionality is a superset of traditional sequential execution paradigms, allowing it to revert to the underlying sequential execution paradigm to handle exceptional circumstances. While this section's discussion is generally kept at a high level (due to space constraints), a few implementation details merit discussion and are covered in Section 4.4. A complete implementation description can be found in [24].

4.1 MSSP Execution Example

We now walk through a detailed MSSP execution example using the code example from Section 3. The flow of the example can be followed in the state diagram shown in Figure 8, The execution (shown in Figure 9) begins partway through the execution of outer loop with the variable `i` equal to 48 and `sometimes_max` equal to 11. For simplicity, we will ignore the other variables. The distilled program is constructed such that the variable `i` is always correctly computed, but the variable `sometimes_max` may be incorrect when the original code executes the inner loop, which has be removed from the distilled program.

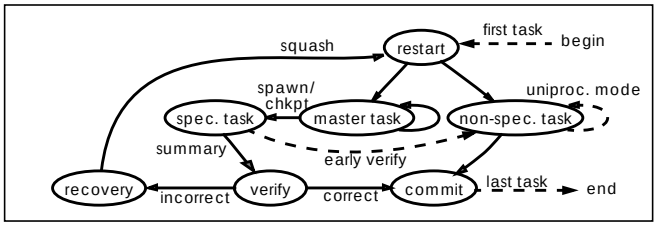

Figure 8. State diagram of MSSP execution

The execution begins with the machine quiesced: all processors are idle and all buffers for non-architected data are empty. The restart process initiates two executions: a non-speculative slave execution and a master execution. Both are assigned the timestamp 7. The first slave can execute non-speculatively because all of its requests for live-in values can be correctly satisfied with architected.

While the non-speculative slave is provided with the architected state, the PC for the master is first mapped (using the mapping hash table) to find the PC of the corresponding entry into the distilled program. The master executes the entry code before entering the distilled program proper. Both tasks execute simultaneously. The non-speculative slave task executes like a normal uniprocessor, immediately committing its state. In this example, we assume that the 16 iterations executed by the non-speculative slave all avoid the inner loop, so its only writes are 16 updates to the variable `i`, of which only the first and last are shown.

Writes performed by the master, on the other hand, are buffered as checkpoint data and associated with the master's current task. In task #7, the master performs a single write (i=64) before encountering a fork instruction. At a fork instruction, the master increments its timestamp and notifies the system that the speculative slave task #8 is ready for execution. This task is allocated to processor *P2*, which was previously idle. When *P2* requests the value `i`, rather than return the current architected value, the memory system sends the checkpoint value (64). Since no checkpoint value is available for `sometimes_max`, the architected value (11) will be returned. The slave processor begins execution in the verify block of the distilled program—using *verifyPC*, provided by the master—which ends with a jump into the original program. While executing the original program, the speculative slave records its live-in values; we will assume that inner loop iterations are executed in both tasks #8 and #9, so the predictions for both variables `i` and `sometimes_max` are recorded and sent to the live-in buffer.

During the slave execution of task #8, the master forks task #9. Since no processor is available, slave task #9 cannot be immediately assigned to a processor. When the non-speculative task on *P0* completes—after its final write to the vari-

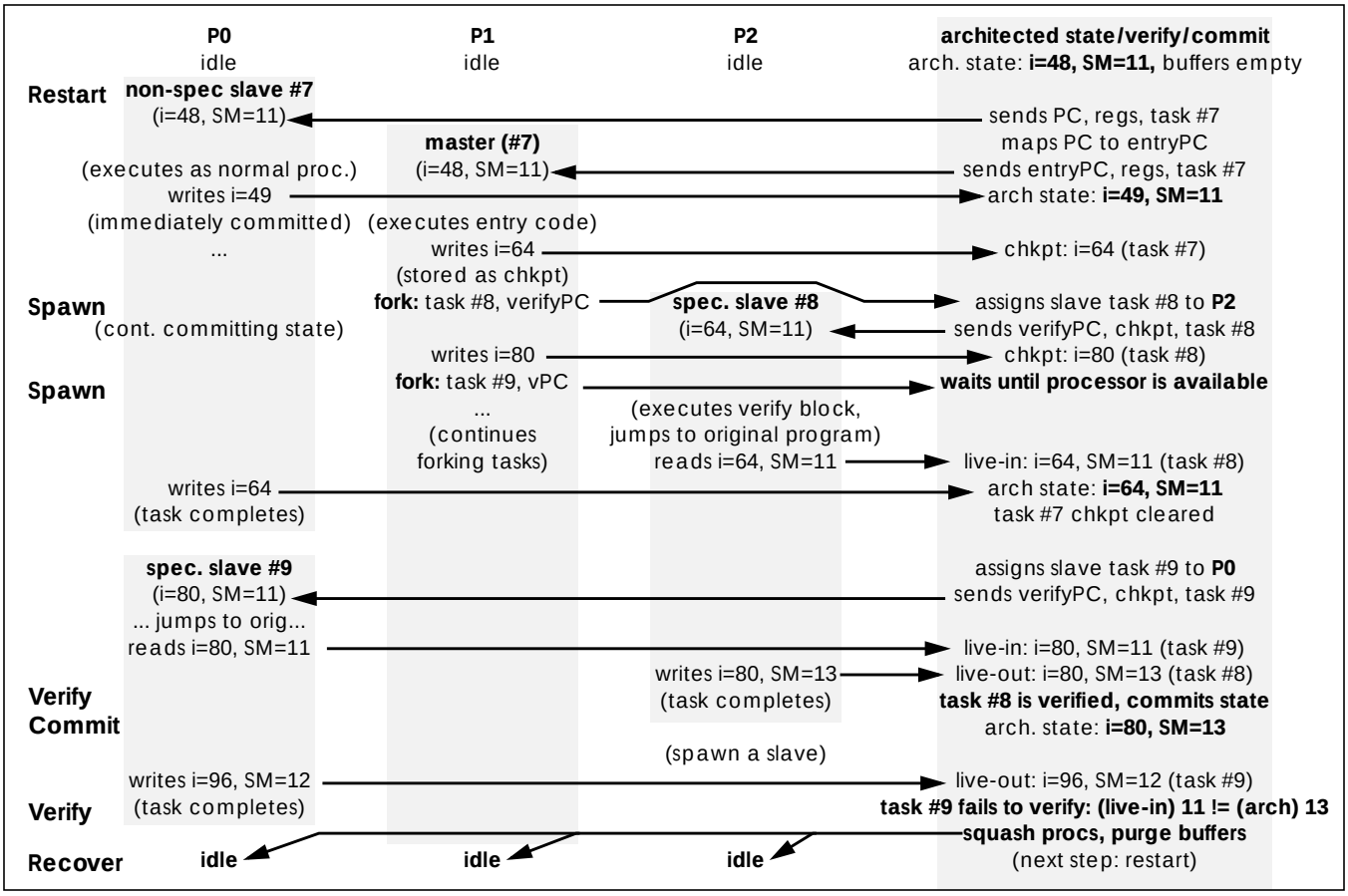

Figure 9. Detailed example MSSP execution: The code example from Figure 5 is executed on 3 processors. The values of only two variables are displayed: `i` (the induction variable of the outer loop, which is correctly updated by the master) and `sometimes_max` (which is abbreviated SM and is incorrectly predicted by the master when the inner loop is executed by the original program). In this example, both speculative slave tasks execute the inner loop, resulting in a task misspeculation when the second task undergoes verification. In the real execution, this situation is uncommon: only one iteration out of every thousand executes the inner loop.

able `i`—task #9 is assigned to *P0*. Speculative slave task #9 is provided with the most recent checkpoint value (80) for `i` and the architected value (11) for `sometimes_max`, because no checkpoint value exists. Like slave task #1, both of these values are read and recorded as live-in values.

At the end of slave task #8, the values for `i` and `sometimes_max` are 80 and 13, respectively. These values are recorded in the live-out buffer. Since task #7 has completed, task #8 is the oldest task in the system and can be verified against architected state. In this case, the live-in values match the architected state, and the live-out values for task #1 can be committed. At this point, the checkpoint and live-in values for task #8 can be discarded.

When task #9 completes and requests verification, a task misspeculation is discovered. When the recorded live-in value for `sometimes_max` (11) is compared to the architected value (13), the values do not match. The live-in predictions the master processor provided to slave task #9 implicitly assumed that task #8 would not modify the variable `sometimes_max`. Since its live-in values were incorrect, the live-out values for task #9 are assumed to be incorrect and discarded. To recover from the task misspeculation, all in-flight tasks are squashed and all non-architected state buffers are purged, returning the system back to the quiesced state.

4.2 MSSP Mechanisms

We describe the MSSP mechanisms in the context of an abstract implementation shown in Figure 10. This abstraction consists of some number of processors and a monolithic memory system that includes infinite buffering for checkpoint and summary data and a task verification/commit unit. Supporting MSSP requires:

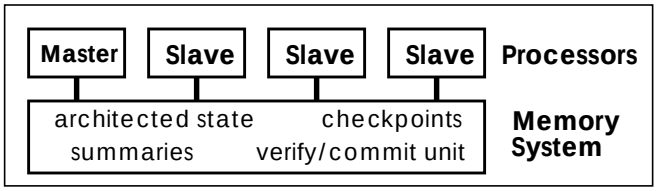

Figure 10. Abstract MSSP implementation with monolithic memory system.

Processor management. A mechanism is required to track idle processors and assign them to slave tasks as the tasks are spawned by the master. To keep data associated with different tasks separate, each task is assigned a sequence number, which we call the *timestamp*, and all non-architected data in the memory system is tagged with its timestamp. When the master executes a fork instruction, it increments its timestamp and the updated timestamp is assigned to the new slave processor.

Checkpoints. Writes performed by the master must be captured to construct the checkpoint differences described in Section 2.2. The collected values are sent to a special structure that is part of the memory system. This structure, the checkpoint buffer, is searched by slave reads. Many small writes from potentially different tasks may need to be merged to satisfy a single read.

Task summarization. Slave processors summarize tasks by collecting the processor's writes (like checkpoint construction) and live-in reads. Tracking reads is much like tracking writes, except that we track only reads to locations not yet written by the task. Thus, the stream of reads is first filtered by the set of preceding writes. Task summaries are stored in the memory system. In addition, slaves must know where their task ends, as described in Section 3.1

Verification and commitment. When a task has completed and all previous tasks have committed, a task can undergo verification. Verification is performed by comparing a task's live-ins to architected state. If they match, architected state can be updated with the task's writes. To avoid memory ordering violations in many consistency models, verification and commit should appear atomic to outside observers. If live-in values do not match architected state, a task misspeculation is signalled and speculative state is flushed, rolling back to the most recent architected state.

Program counter mapping. Since the distilled program resides at a different location in memory than the original program, entry and indirect branch target PC's need to be mapped using the tables generated by the distiller (see Section 3.5). These lookups exhibit locality that can be exploited by a TLB-like mechanism.

4.3 Reverting to Sequential Execution

To handle extraordinary circumstances, MSSP falls back on sequential execution. At any point, all speculative state can be purged, and a single processor can be assigned to execute the program sequentially, starting from the last committed architected state. Such a technique is used in many speculative parallel architectures. In addition, the rePLay framework [9] similarly falls back on traditional execution when its speculatively optimized code is inappropriate.

MSSP uses this technique for three purposes: exceptions, very large tasks, and to ensure forward progress. Upon detecting an exception, the execution can be rolled back to architected state. If the exception occurs again in a sequential execution, it can be handled by one of the existing techniques for handling precise exceptions. Very large tasks may exceed the amount of buffering available for task summarization; if this occurs, the task can be completed through sequential execution, which requires no such buffering. MSSP can fail to make forward progress due problems introduced into the distilled program by inappropriate approximations (*e.g.*, infinite loops, etc.). We include a watch dog timer to detect when deadlock/live-lock occurs and signal a recovery. The execution can proceed sequentially until it is deemed safe to return to MSSP mode.

We apply an optimization called early verification to avoid squashing the portion of the task that has been completed when reverting to sequential execution. To perform *early verification*, a slave processor selects an arbitrary instruction at which to end the task summary. The summary can then be verified; if committed, the architected state will reflect the state at that stopping point, and the task execution can continue non-speculatively.

4.4 Implementation Details

Of MSSP's mechanisms, two are most critical for performance: collection of checkpoint data and verification/commit. As the execution of the distilled program will likely determine performance, the collection of checkpoint writes (both register and memory) should be done in a way that minimally impacts the master's execution throughput.

To ensure verification/commit has sufficient throughput, it will likely be necessary to bank the level of the memory system at which the verification/commit unit resides. Such an arrangement requires a two-phase commit protocol to assure all banks agree to verification before any data is committed. A two-phase commit protocol need not introduce a sequential communication latency, because each bank can speculatively verify (*i.e.*, perform verification assuming all previous tasks commit) its portion of the task summary by itself. This allows the two-phase commit protocol for separate tasks to be pipelined.

Similarly, it will likely be necessary to cache checkpoint data at the processors to avoid accessing the shared-level of the memory hierarchy on every access. As the checkpoint data is not architected state, it must be invalidated appropriately. At the end of tasks, slave processors must invalidate cache lines that contain checkpoint data or have been updated by task commitment. As a performance enhancement, the master should periodically update cache lines containing old non-architectural data, so that more recently committed values are visible. This is done in our implementation via *refreshing*: the master periodically requests an up-to-date copy of cache lines already present in its data cache.

5 Experimental Results

In this section, we describe experiments performed to quantify the potential of the MSSP paradigm. As with any technique that involves transforming programs, performance potential is difficult to quantify indirectly (*i.e.*, without building the system that actually transforms the program). As a result, we have implemented a first-cut automatic program distiller (described in Section 5.1) to allow us to quickly explore some of the characteristics of MSSP execution. In no way are these results a "limit study" of the performance of MSSP.

Due to space constraints we present the results of a selection of experiments in Section 5.2. The complete evaluation can be found in [24]. Our initial experiments have demonstrated that the MSSP paradigm has a number of desirable characteristics:

1. Code approximation can be used to generate accurate value predictors for task live-in values. On average, task misspeculation occurs only once every 10,000 to 200,000 original program instructions.
2. MSSP performance is largely a function of the effectiveness of distillation, which varies among programs. Speedups (over a single processor) range from negligible speedup (`gzip`) to 1.70 (`gcc`, `vortex`), with a harmonic mean of 1.25. We believe there is much headroom to improve distillation.
3. Accurate predictions make MSSP largely insensitive to inter-processor communication: as inter-processor communication latency is scaled from 5 to 20 cycles, relative performance only degrades by 10 percent.
4. Only a modest amount of speculative buffering is required: for these experiments, the primary storage of checkpoint, live-in, and live-out data (at the L2) needed only 24kB of storage total: about 1% of the storage of a 2MB L2 cache.

5.1 Experimental Methodology

Our simulation-based evaluation involved three components: 1) a prototype automatic program distiller, 2) an execution-based timing simulator, and 3) the SPEC2000 integer benchmarks.

Although program distillation may be most fruitful if performed at run time, for simplicity our first implementation performs distillation statically and off line. The distiller is like a binary-to-binary translator: it augments an Alpha architecture binary with a distilled program and the requisite mapping tables. Currently, the distiller performs approximations with the parameters listed in Table 1. These transformations are guided by profile information; to approximate the quality of profile information available to a run-time implementation, profile information was gathered from the same run of the program with which the evaluation was performed. After approximation, the following traditional optimizations were applied: dead code elimination, limited constant folding, register (re-)allocation, in-lining, and profile-guided code layout.

Our simulation model is derived from the SimpleScalar toolkit [4], but the timing model was re-written from scratch. In our model, timing and functional simulation are not decoupled; instead, great care is taken to simulate operations as and when the simulated machine performs them (*e.g.*, registers are renamed and misspeculation recovery is performed by rewinding to a pre-misspeculation register map). Our simulated machine is an eight-processor CMP, where each processor is configured to be similar to the Alpha 21264 [14]. Processor parameters are shown in Table 2. Uncontended latency on the interconnect is 10 cycles, so the minimum L2 hit latency is 26 cycles (round-trip on the interconnect + cache access time).

We used the SPEC2000 integer benchmarks, because they exhibit the control flow and memory access behaviors representative of non-numeric programs. The benchmarks were

Table 1. Parameters supplied to the program distiller.

Parameter	Value	Explanation
Target Task Size	250 insts	The task size the task selection algorithm considers optimal.
Branch Threshold	99% biased	Fraction of instances that must go to the dominant target for the branch to be removed.
Idempotency Threshold	99% correct	Fraction of times that an instructions output must match an input for it to be removed.
Distant Use Definition	1000 insts	Store to load distance at which a dependence is considered as having a distant first use.
Distant Use Threshold	99% distant	Fraction of store instances that have to be distant for the static store to be considered distant.

Table 2. Simulation parameters approximating a CMP of Alpha 21264 processors.

Front End	A 64KB 2-way set associative instruction cache, a 64Kb YAGS branch predictor, a 32Kb cascading indirect branch predictor, and a 64-entry return address stack. The front end fetches aligned blocks of 4 instructions. A line-predictor à la the 21264 is implemented.
Execution Core	4-wide (fetch, decode, execute, retire) machine with a 128-entry instruction window and a 13 cycle pipeline, includes a full complement of simple integer units, 2 load/store ports, and 1 complex integer unit, all fully pipelined. The L1 data cache is a 2-way set-associative 64KB cache with 64B lines and a 3-cycle access latency, including address generation.
System	The system on a chip includes 8 processor and 8 L2 cache banks (each 4-way set-associative with 64B lines, 2MB total), and a register bank. An interconnection network is modelled that can move one 64B message per cycle to or from each processor and cache; one-way latency is 10 cycles. Minimum memory latency (after an L2 miss) is 100 cycles.

compiled for the 21264 implementation with the "peak" optimizations specified. As such, the binaries contain many nops for branch alignment, which benefit the 21264-style line-predictor-based front end we model; these nops are fetched but discarded after decode. We used modified reference input sets allowing us to run the benchmarks to completion (9 - 45 billion instructions). The input sets were chosen to maintain the memory working sets of the reference inputs.

5.2 Results

For these experiments, we request our task selection algorithm to create tasks that execute 250 dynamic instructions in the original program. The resulting distribution of task sizes (weighted by execution frequency) is shown in Figure 11. Although the average task size is closer to 200 instructions, most tasks are between 100 and 300 instructions. Tasks of this size appear to be a good compromise between storage requirements (which increase with task size) and inter-processor bandwidth usage (per-instruction bandwidth usage decreases with larger tasks).

Figure 12a shows the average distance (in original program instructions) between task misspeculations. This distance is generally above 10,000 instructions and as high as 200,000 instructions, demonstrating that distilled programs can be used as highly accurate predictors. The effectiveness of distillation that corresponds to this misspeculation rate is shown in Figure 12b. This plot shows the *distillation ratio*, the ratio of distilled to original program instructions executed, not counting nops. Currently, our distiller is most effective (lower is better) on two benchmarks (`gcc` and `vortex`), where the master executes two thirds fewer instructions than the original program. Although some programs may be intrinsically easier to distill than others, we believe the variation in distillation effectiveness may be a consequence of the selection of optimizations currently implemented in our distiller prototype.

The distillation ratio correlates well with the speedup that MSSP achieves over a traditional uniprocessor execution on the same simulated hardware (shown in Figure 12c). Many of the speedups are in the 1.2 range, but `gcc` and `vortex` demonstrate the potential of the MSSP paradigm, achieving speedups of 1.7. The average number of original program

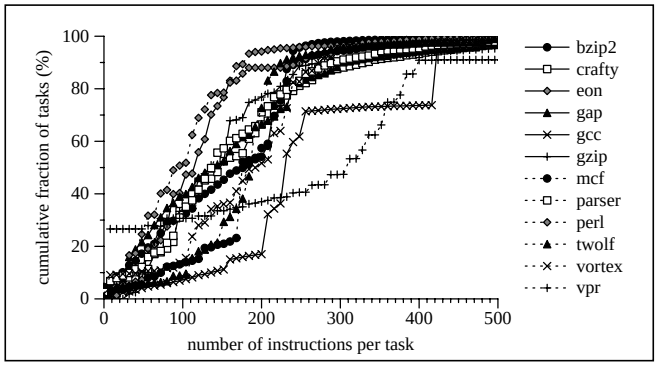

Figure 11. Distribution of task sizes.

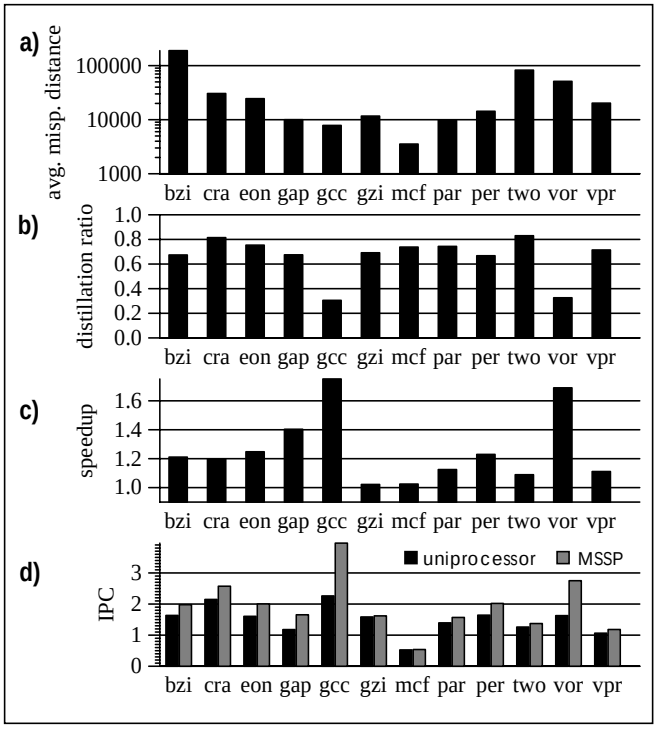

Figure 12. Performance of MSSP execution.

instructions retired per cycle is shown in Figure 12d. In some phases `gcc`'s aggregate IPC exceeds the width of a single core.

The simulations modeled an eight processor CMP, but most of the benchmarks rarely use more than five processors (shown in Figure 13). The average number of in-flight tasks is slight larger, as processors can be re-allocated to new tasks upon completion of a task, not when the task is committed.

In what follows, we summarize results from [24]. The storage requirements for checkpoint, live-in, and live-out values can be modest. Checkpoints are biased toward register writes: for average 200 instruction tasks, the master writes 15 registers and seven distinct 8-byte memory words. In contrast, live-in sets are dominated by loads, with 20 distinct 8-byte memory words read but only six registers. Live-out

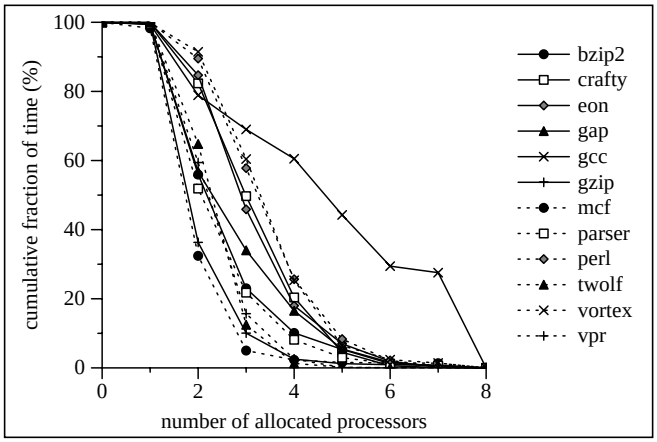

Figure 13. Distribution of processor usage.

sets are strictly larger than checkpoints, consisting of 20 register writes and 12 distinct 8-byte memory words written. For these benchmarks and this implementation, the total storage required at the L2 for special buffers is around 24kB, a small amount of storage in a next generation processor.

Sensitivity analysis leads to the following observations: 1) exploiting the enabled safe optimizations is essential, as they are responsible for two thirds of the benefit of distillation and generally affect the misspeculation rate negligibly, 2) MSSP execution is largely tolerant of interconnect latency: when scaled from 5 to 20 cycles, speedup relative to a uniprocessor execution only decreases 10 percent, 3) benchmarks that can keep many tasks in flight simultaneously (*e.g.*, `gcc`) perform significantly worse with fewer processors; others achieve similar performance of four processors as they do on eight,4) our current distillation infrastructure is insensitive to the task boundaries selected and correctness thresholds used in the approximation transformations, as long as tasks are not too small (at least 100 instructions on average) and important optimizations (*e.g.*, unrolling of important loops) are not prevented.

6 Conclusion

In this paper, we presented a novel execution paradigm, Master/Slave Speculative Parallelization (MSSP), that dedicates one processor, the master, to predicting the live-in values required by the remaining processors. A key component of MSSP is the distilled program, a speculative approximation of the original program. Since the distilled program has no correctness constraints, it circumvents the need for optimizations to be correct, a requirement that hampers traditional optimizing compilers.

We believe that MSSP conforms to the necessary real-world constraints to become widely adopted; in particular, it can be transparent, require modest verification effort beyond a traditional CMP, and conform to silicon technology trends. Since the original program is used un-modified, there are no required compiler changes and legacy binaries can be supported. The distilled code, which can be derived from the original program, has no correctness requirements. As a result, the program distiller need not be verified. The architecture itself is tolerant of wire latency, because inter-processor communication is only on the critical path when the master misspeculates, an occurrence that our results show is infrequent.

Acknowledgements

We would like to thank Ras Bodik, Adam Butts, Joel Emer, Konrad Lai, Milo Martin, C. J. Newburn, Dan Sorin and the anonymous reviewers for their contributions to the content and presentation of this paper. This work was supported in part by National Science Foundation grants CCR-9900584 and EIA-0071924. Craig Zilles was supported by a Wisconsin Distinguished Graduate Fellowship.

References

[1] H. Akkary and M. Driscoll. A Dynamic Multithreading Processor. *Micro-31*, Nov. 1998.

[2] T. Austin. DIVA: A Reliable Substrate for Deep Submicron Microarchitecture Design. *Micro-32*, Nov. 1999.

[3] V. Bala, E. Duesterwald, and S. Banerjia. Transparent Dynamic Optimization. Technical Report HPL-1999-77, Hewlett Packard Labs, June 1999.

[4] D. Burger and T. Austin. The SimpleScalar Tool Set, Version 2.0. Technical Report CS-TR-1997-1342, Computer Sciences Department, University of Wisconsin–Madison, 1997.

[5] B. Calder, P. Feller, and A. Eustace. Value Profiling and Optimization. *Journal of Instruction Level Parallelism*, Mar. 1999.

[6] M. Cintra, J. Martinez, and J. Torrellas. Architectural Support for Scalable Speculative Parallelization in Shared-Memory Systems. *ISCA-27*, June 2000.

[7] J. Collins, *et al*. Speculative precomputation: Long-range prefetching of delinquent loads. *ISCA-28*, July 2001.

[8] P. Dubey, *et al*. Single-Program Speculative Multithreading (SPSM) Architecture: Compiler-Assisted Fine-Grained Multithreading. *PACT*, June 1995.

[9] B. Fahs, *et al*. Performance Characterization of a Hardware Framework for Dynamic Optimization. *Micro-34*, Dec. 2001.

[10] J. Fisher. Trace scheduling: a technique for global microcode compaction. *IEEE Transactions on Computers*, 30(7):478–490, 1981.

[11] D. Gallagher, *et al*. Dynamic memory disambiguation using the memory conflict buffer. *ASPLOS-6*, Oct. 1994.

[12] L. Hammond, M. Willey, and K. Olukotun. Data Speculation Support for a Chip Multiprocessor. *ASPLOS-8*, Oct. 1998.

[13] W. M. Hwu, *et al*. The Superblock: An Effective Technique for VLIW and Superscalar Compilation. *Journal of Supercomputing*, 7(1):229–248, Mar 1993.

[14] R. Kessler. The Alpha 21264 Microprocessor. *IEEE Micro*, 19(2):24–36, March/April 1999.

[15] P. Marcuello, J. Tubella, and A. Gonzalez. Value Prediction for Speculative Multithreaded Architectures. *Micro-32*, Nov. 1999.

[16] J. Oplinger, D. Heine, and M. Lam. In Search of Speculative Thread-Level Parallelism. *PACT*, Oct. 1999.

[17] E. Rotenberg, *et al*. Trace Processors. *Micro-30*, Dec. 1997.

[18] A. Roth and G. Sohi. Speculative Data-Driven Multi-Threading. *HPCA-7*, Jan. 2001.

[19] A. Sodani and G. Sohi. Dynamic Instruction Reuse. *ISCA-24*, June 1997.

[20] G. Sohi, S. Breach, and T. Vijaykumar. Multiscalar Processors. *ISCA-22*, June 1995.

[21] J. Steffan, *et al*. Improving Value Communication for Thread-Level Speculation. *HPCA-6*, Jan. 2000.

[22] K. Sundaramoorthy, Z. Purser, and E. Rotenberg. Slipstream Processors: Improving both Performance and Fault Tolerance. *ASPLOS-9*, Nov. 2000.

[23] J.-Y. Tsai and P.-C. Yew. The Superthreaded Architecture: Thread Pipelining with Run-Time Data Dependence Checking and Control Speculation. *PACT*, Oct. 1996.

[24] C. Zilles. *Master/Slave Speculative Parallelization and Approximate Code*. PhD thesis, Computer Sciences Department, University of Wisconsin–Madison, Aug. 2002.

[25] C. Zilles and G. Sohi. Execution-based Prediction Using Speculative Slices. *ISCA-28*, July 2001.