Master/Slave Speculative Parallelization with Distilled Programs

Craig B. Zilles and Gurindar S. Sohi
[zilles, sohi]@cs.wisc.edu

Abstract

Speculative multithreading holds the potential to substantially improve the execution performance of sequential programs by leveraging the resources of multiple execution contexts (e.g., processors or threads). For unstructured non-numeric programs, the three key challenges of parallelization are (1) predicting the sequence of tasks (i.e., groups of instructions) that corresponds to the correct sequential execution of the program, (2) providing each task with the values it needs to execute (i.e., its live-in values), and (3) tolerating the communication latency between processors.

To overcome these challenges, we propose a new execution model that differs from previous speculative multithreading models because of its **master/slave** nature. In this model, one execution context—the master—executes a speculative approximation of the original program—the **distilled program—**that allows it to anticipate future control- and data-flow and explicitly orchestrate the parallel execution. Because the control flow in the two programs roughly corresponds, the master can accurately predict the sequence of tasks by mapping its program counter (PC) in the distilled program to a task start PC in the original program. Furthermore, speculative state (e.g., register and memory values) generated by the execution of the distilled program serves as predicted live-in values for the tasks. These predictions reduce the impact of communication latency on execution performance.

We present an analytical model that shows that, if predictions made by the master are accurate, the execution performance closely tracks that of the distilled program. The distilled program can execute faster than the original program for two reasons: (1) it generates only a subset of the state generated by the original program, and (2) it need not perform the predictable portion of the computation, because the predictions will be verified by the parallelized execution of the original program. We perform an initial exploration of the potential of distilled programs, showing that dynamic instruction count can be reduced significantly with minimal impact on accuracy.

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.

Parallelization without programmer intervention can be achieved by analyzing the program's dependences, partitioning the program into independent subsets, and inserting the necessary synchronization. For non-numeric programs, a complete dependence analysis is difficult. As a result, parallelization for these programs is greatly facilitated by speculating in the presence of ambiguous dependences and provid-

University of Wisconsin-Madison Technical Report 1438. Available from http://www.cs.wisc.edu/~zilles/papers/mssp-tr1438.pdf

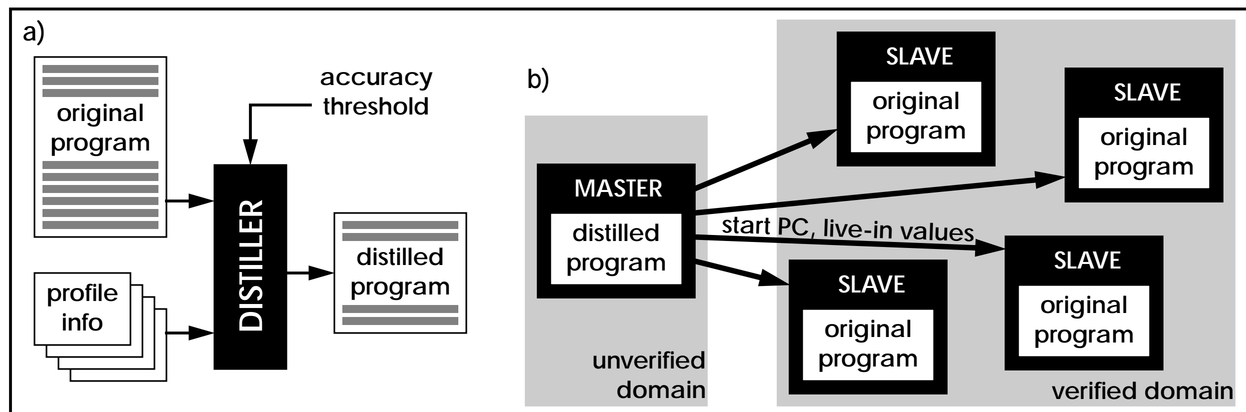

Figure 1. Program distillation and master/slave speculative parallelization. The program distiller (a), which could be hardware or software, takes the original program, profile information, and a tunable accuracy parameter to generate the distilled program. During execution (b), the master executes the distilled program to predict starting PCs and live-in values for the slave processors. The slaves verify that the live-in predictions correspond to a sequential execution. The master is not explicitly verified, but restarted whenever the slaves detect a misspeculation.

ing hardware support for detection of and recovery from actions that violate the ordering dictated by a sequential execution.

In this paper, we present a new speculative parallelization execution model based on a master/slave architecture. Our new execution model is motivated by the repetition and predictability exhibited by programs. It is well understood that there are many aspects of a program's execution that are trivially predictable (*e.g.,* many static branches are only ever taken in one direction). Despite this predictability, the compiler cannot safely remove the predictable behavior, even with accurate profile information, because the past is not a guarantee of future behavior.

We propose generating a speculative approximation of the program—a second static image we call the *distilled program*—in which the predictable code has been removed. Removal of predictable code reduces dynamic instruction count and exposes new opportunities for traditional compiler optimizations. The execution of the distilled program should significantly outperform, while closely paralleling in function, the original program, but it provides no correctness guarantees. In fact, a spectrum of distilled programs exists. As shown in Figure 1(a), an accuracy parameter is specified to the program distiller. Lowering this parameter typically enables the distiller to produce code that is faster in the common case but misspeculates more frequently.

In our proposed execution model, one processor[1]—the *master*—executes the distilled program to produce accurate predictions of the program's future behavior. The master uses these predictions to orchestrate a parallel execution of the original program on the remaining *slave* processors (see Figure 1(b)). The master provides each slave processor with a starting program counter (PC) and predictions for the live-in

1. For simplicity of exposition we use the word "processor", but the ideas are equally applicable to other execution contexts (*e.g.,* SMT threads).

University of Wisconsin-Madison Technical Report 1438. Available from http://www.cs.wisc.edu/~zilles/papers/mssp-tr1438.pdf

values the slave requires. The start PC is produced by mapping the master's PC in the distilled program to the corresponding location in the original program. The live-in predictions are derived from values computed by the execution of the distilled program.

The slave processors execute the un-modified original program (*i.e.*, compiler modifications, while potentially beneficial, are not required), and only they are allowed to affect architected state. Values computed by the master are buffered while they are needed as live-in predictions and then discarded. Because all communication between the master and the slaves is in the form of predictions, which are verified before the slave updates architected state, there are absolutely no correctness requirements on the distilled program. This facilitates constructing the distilled program at run-time, when the most accurate profile information is likely to be available.

The two executions must run in close succession—the master's lead is limited by the availability of speculative buffering—so performance will be determined by the slower of the two. To achieve execution throughput equivalent to the master, additional processors can be allocated to the slave execution. If the distilled program is capable of outperforming the original program by a factor of **N**, then **N** slave processors are allocated to "match the impedances" of the two executions. The resource overhead (**1/N**) of dedicating a master processor to orchestrate the execution decreases as the degree of parallelization increases.

The organization of the paper is as follows: in Section 2, we present our execution model, comparing it with previous speculative parallelization models. Section 3 describes a possible implementation of the model, providing detail on how the master "forks" tasks and provides live-in state and how the master is restarted following a misspeculation. In Section 4, a simple analytical model is proposed to allow reasoning about performance. In Section 5, we perform an initial exploration of distilled programs, analyzing the effectiveness of a variety of speculative optimizations. We describe related work before concluding.

2 The Master/Slave Speculative Parallelization (MSSP) Execution Model

Two key components of any speculative parallelization model are predicting the sequence of *tasks* (continuous segments of the dynamic instruction stream) and handling inter-task communication. Although partially performed in software, the task sequencing performed by MSSP is in principle equivalent to having a centralized next task predictor. The central difference between MSSP and previous speculative parallelization models is in inter-task data communication. In this section, we describe how using a master processor improves the communication and computation of live-in values, how live-ins can be verified, and how distilled programs enable the master processor to perform its role.

2.1 Optimizing Live-In Communication

The values a task reads that it did not generate itself are the task's *live-in values*. For a moment, let us consider a simple example (shown in Figure 2(a)) in which each task is a loop iteration and the loop

University of Wisconsin-Madison Technical Report 1438. Available from http://www.cs.wisc.edu/~zilles/papers/mssp-tr1438.pdf

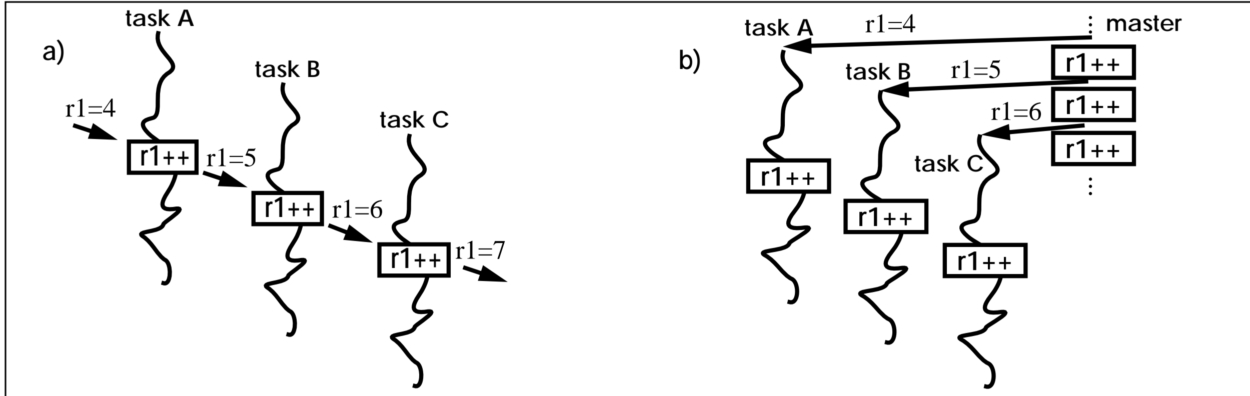

Figure 2. Inter-task communication in speculative parallel architectures: An illustrative example, where the only inter-task dependence is due to a register allocated counter that is incremented by each task. (a) In previously proposed speculative parallelism architectures, the live-ins are supplied from other in-flight tasks. (b) In MSSP, the master processor predicts task live-in values.

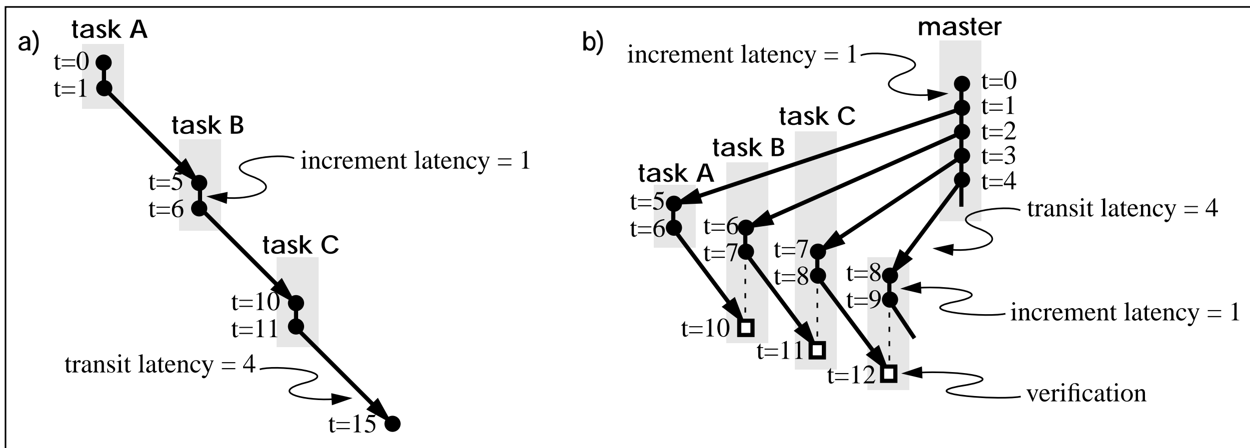

Figure 3. Critical paths through speculatively parallel executions: (a) With the traditional approach, the critical path includes one transit delay for each task, leading to a maximum rate of one task every 5 cycles. (b) With the master delivering live-in values, transit latency is incurred in parallel, leading to a maximum rate of one task per cycle, the increment latency.

index—incremented by each task—constitutes the only inter-task communication. Previous speculative parallelization techniques [1, 3, 7, 10, 12, 16, 18, 20, 30, 29, 32] provide this live-in state from the task that computed it. The execution's critical path (shown in Figure 3(a)) consists of two components: 1) the computation to convert the live-in value to the next task's live-in value, and 2) the transit time for the value to be passed from one task to the next. If we assume inter-task transit time of 4 cycles and a single cycle increment, we can execute tasks at a rate of no more than one every 5 cycles.

To avoid sequential transit delays, MSSP provides the live-in values from a 3rd party, the master (as shown in Figure 2(b)). The master executes only the code that makes up the task live-ins, in this case only the register increments. The new critical path is shown in Figure 3(b). Because the transit delays are now in parallel, we can potentially achieve a task throughput of one per cycle (*i.e.*, the latency of the increment instruction).

University of Wisconsin-Madison Technical Report 1438. Available from http://www.cs.wisc.edu/~zilles/papers/mssp-tr1438.pdf

2.2 Optimizing Live-In Computation

In our simple example, the computation component was a single cycle. In general, inter-task communication can consist of any number of values and can be the result of arbitrary computation. Previous speculative parallelization schemes try to minimize the computation component through code scheduling by moving definitions of live-outs up and uses of live-ins down [30, 33]. This code motion must be proven correct by the compiler or misspeculation detection and recovery code is necessary. Optimizations of this sort create a tension between generating 100% correct code and computing future live-ins as quickly as possible.

MSSP resolves this tension by decomposing the program into two programs. Live-in values computed by the master are treated as value predictions, allowing the master's program—the distilled program—to be optimized without fear of violating correctness. Conversely, with reduced performance requirements, the slave's code can focus on correctness. Live-in values are verified to detect incorrect predictions.

2.3 Verification

As with any speculative parallelism architecture, it is necessary to verify that stitching the tasks together results in the original sequential execution. One means of accomplishing this verification is to buffer the speculatively-used live-in values and compare them to the architected state generated by the previous task at retirement. As shown in Figure 4, we record in the *live-in buffer* the name (*e.g.,* architected register or address) and value for each live-in consumed by the task. Values produced within the task need not be stored, as they can be proven correct transitively. When the previous task is complete and has successfully updated architected state, we compare the values stored in the live-in buffer to the architected state. If all values match, then the task has been verified and is free to update architected state[2]. This is equivalent to the reuse test [28] but is performed at the task granularity.

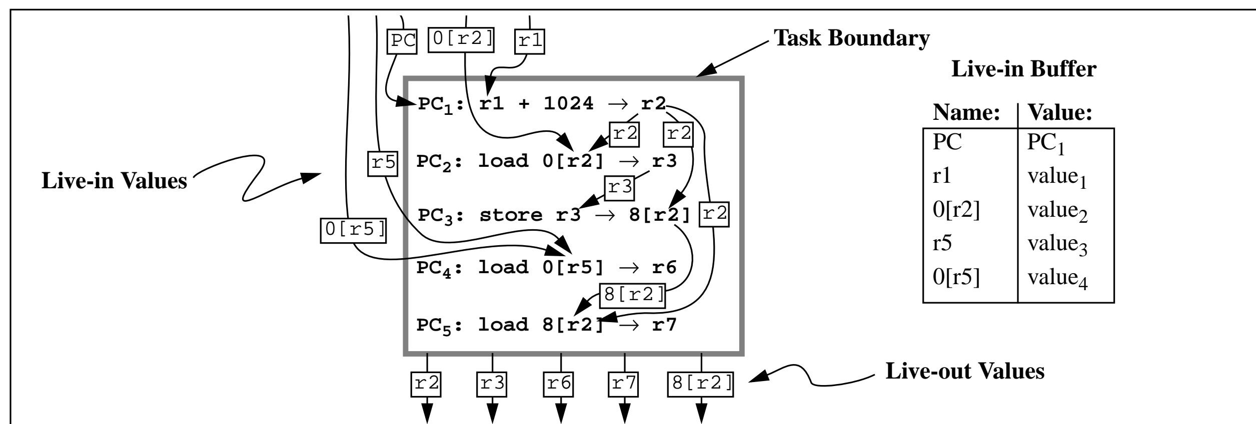

Figure 4. Live-in value verification. Name-value pairs of live-ins are stored in the live-in buffer for verification when previous task has retired. Sourced values produced within the task (*e.g.*, r3) need not be verified. Once the task has been verified, architected state can be updated with live-out values.

University of Wisconsin-Madison Technical Report 1438. Available from http://www.cs.wisc.edu/~zilles/papers/mssp-tr1438.pdf

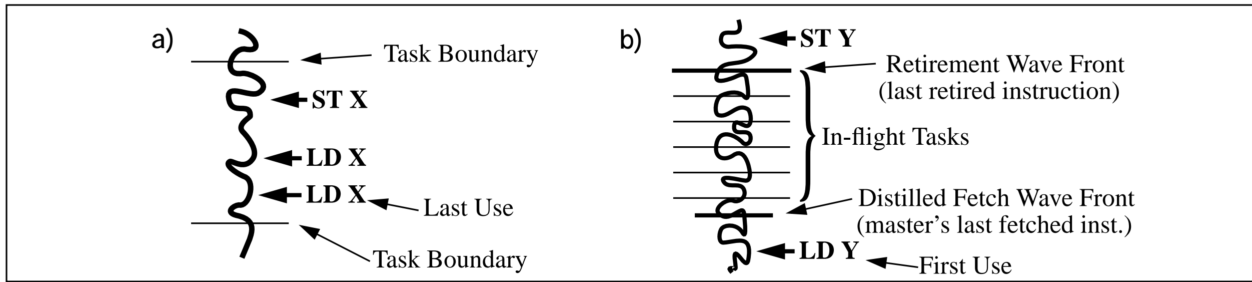

Figure 5. State updates that need not be included in checkpoints produced by the distilled program. (a) If the definition and all uses of a value occur within a task, it will not be a live-in. (b) If the definition and first use of a value are not simultaneously in-flight (*i.e.*, the definition is retired before the first use is fetched), then the live-in value can be retrieved from architected state.

2.4 Distilled Programs

The code executed by the master is expected to predict (1) the sequence of tasks and (2) their live-in values. If the master executed the original program, it could "predict" task start PCs by using hardware to monitor the execution and periodically (*e.g.*, every 100th retired instruction) sending a slave processor its current PC. In addition, the master would have already computed the complete memory and register image associated with the beginning of the task and could provide the slave with any desired live-in value. Thus, the original program could be used as a perfect predictor for start PCs and live-ins, but it is unlikely to outperform a sequential execution.

To allow us to derive benefit from the parallelization, we *distill* the original program to the minimal computation that fulfills the master's requirements. One component of distillation, as mentioned in Section 1, is removing predictable behaviors from the distilled program. In addition, the distilled program need only compute a subset of the original program's state.

Only values that frequently form a task's live-in state need to be computed by the distilled program. Values that are created and killed within a task (shown in Figure 5(a)) can never be a live-in value and therefore may not need to be computed. Furthermore, because the parallelized tasks will eventually perform all stores and register writes in the original program, the distilled program need only compute values whose definition and use could be in-flight simultaneously. If the distance (in dynamic instructions in the original execution) between the definition and the first use is large (as shown in Figure 5(b)), the definition need not be in the distilled program, because the value will be available from the architected state by the time the use is executed.

2. For memory consistency in multiprocessors it is necessary to monitor the addresses of *all* loads for coherence requests between verification and retirement. Alternatively, the reuse test and architected state update can potentially be performed atomically with respect to the memory system by acquiring the necessary coherence permissions for blocks containing live-in and live-out values, as is described in [23].

University of Wisconsin-Madison Technical Report 1438. Available from http://www.cs.wisc.edu/~zilles/papers/mssp-tr1438.pdf

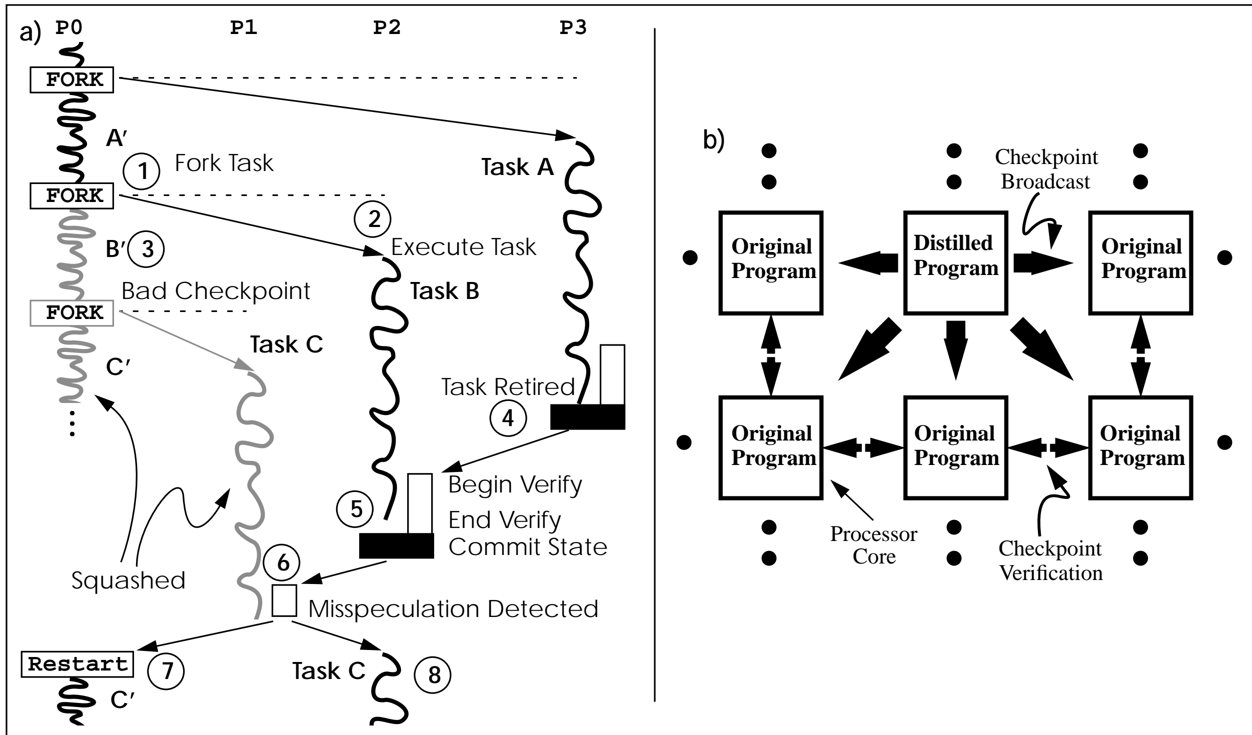

Figure 6. Master processor distributes checkpoints to slaves. (a) The master, executing the distilled program on processor P0, forks tasks, providing them live-in values in the form of checkpoints. The live-in values are verified when the previous task retires. Misspeculations, due to incorrect checkpoints, cause the master to be restarted with the architected state. (b) Physically, this architecture can be mapped onto a chip multiprocessor.

3 Possible Implementation

In this section, we describe an example of an MSSP architecture to discuss the types of mechanism that are required. A full study of this architecture is beyond the scope of this paper and is left for future work. We first present an overview of the architecture, then describe how the checkpoints work, and finally discuss the misspeculation detection and recovery path.

3.1 Execution Model

The master (**P0** in Figure 6(a)) manages the parallel execution through explicit *fork instructions* present in the distilled program. Upon encountering a fork instruction, **P0** spawns (**1**) the next task (**Task B**) in the original program on a free processor (**P2**), providing it with a "checkpoint" of the master's current state (checkpoints are described in detail in Section 3.2). After some latency, the checkpoint arrives and **P2** can begin executing the task (**2**). **P0** continues executing (**3**) the distilled program segment that corresponds to task B, which we refer to as **B'**.

As task B executes, it retrieves its live-ins from the checkpoint, recording the name-value pairs that it consumes. When the previous task (**Task A** on **P3**) is complete (**4**), **P2** can begin checking its live-in values against the architected state. If the recorded live-in values exactly correspond to architected state, then the

University of Wisconsin-Madison Technical Report 1438. Available from http://www.cs.wisc.edu/~zilles/papers/mssp-tr1438.pdf

task has been verified and can be retired, and architected state can be updated (**5**) with the task's live-out values.

If a checkpoint contains an incorrect value (**3**) for a live-in—because the distilled program produced the wrong value or did not produce a needed value—this 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, execution of the corresponding task in the original program (**Task C**) begins.

The MSSP execution model is easily mapped onto an explicitly parallel architecture, such as the CMP shown in Figure 6(b). Existing CMP designs (*e.g.,* dynamic superscalar or EPIC processors connected with a high bandwidth, low latency interconnection network) could be enhanced to support the management of checkpoint state, detection of and recovery from live-in value misspeculations, and mapping between the distilled and original programs. We discuss the requirements of each of these enhancements the next three sections.

3.2 Checkpoints

The checkpoints, distributed to provide task live-in values, cannot consist of a complete copy of the program's memory image, nor is such a copy necessary. The checkpoint needs only include name-value pairs for which the checkpoint differs from architected state. As is shown in Figure 7(a), each segment of the distilled program's execution between fork instructions produces a partial checkpoint of the values created by the segment. The master records the name-value pairs and tags them with a sequence number called the *partial checkpoint number.*

The complete state image required by a task is provided by the un-retired partial checkpoints (ordered from youngest to oldest) together with the architected state. When a live-in value is required, the slave processor logically accesses each partial checkpoint in sequence, looking for the first value with a matching name (shown in Figure 7(b)). If no match is found, the architected value is used. This process is very similar to what is required for other speculative parallelization techniques. To avoid increasing cache access latency by sequentially accessing partial checkpoints, the task's view of the block can be assembled at cache fill time, as was previously proposed in [15, 16, 30]. Each partial checkpoint can be deallocated

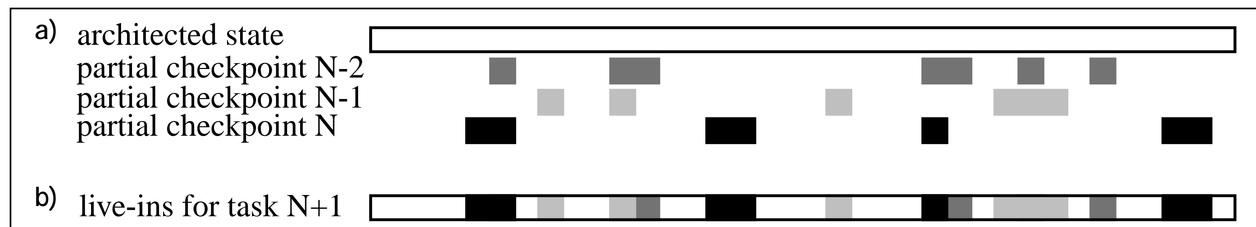

Figure 7. Live-in checkpoint is assembled from partial checkpoints. (a) Each segment of the distilled program's execution produces a partial checkpoint. (b) A checkpoint image for task N+1 is assembled by selecting the most recent copy of each value from the partial checkpoints and architected state.

University of Wisconsin-Madison Technical Report 1438. Available from http://www.cs.wisc.edu/~zilles/papers/mssp-tr1438.pdf

when the corresponding task completes its update of architected state. At this point, the architected state reflects the complete and correct execution of the task, and the partial checkpoint is no longer necessary.

As discussed in Section 3.1, the generation of checkpoints is dictated by special fork instructions in the distilled program (shown in Figure 8). These fork instructions have an instruction format like an unconditional direct branch, but, when executed, two paths are followed. The master continues executing the fall-through path, and the branch target path is spawned on an idle slave processor. If no idle processors are available, the spawn can be buffered or the distilled program stalled. The fork instruction also increments the partial checkpoint number.

To allow cross-task optimizations in the distilled program, the slave processor executes transition code before branching to the original program (see Figure 8). Without the transition code, the distilled program's state would have to correspond exactly to that of the original program at every fork instruction. The transition code enables optimizations like re-allocating registers and hoisting code across checkpoints by restoring the state expected by the original program. By executing the transition code on the slave processor, we minimize the work performed by the master. Transition code only updates the local copy of the checkpoint (*i.e.*, it does not update architected state and therefore need not be tracked for verification purposes).

The checkpoint state includes a starting PC (discussed in more detail in the next two sections) but provides no indication of where the task should end. Intuitively, each task should end where the next task begins, so they can be stitched together to make a complete, non-redundant execution. To this end, we statically annotate each original program instruction (in the form of a bitmap that parallels the original program's static image) with whether the instruction corresponds with the beginning of a task, much like Multiscalar's *stop bits* [29]. As each task executes, it checks whether this checkpoint bit is set, and stops when a set bit is encountered[3].

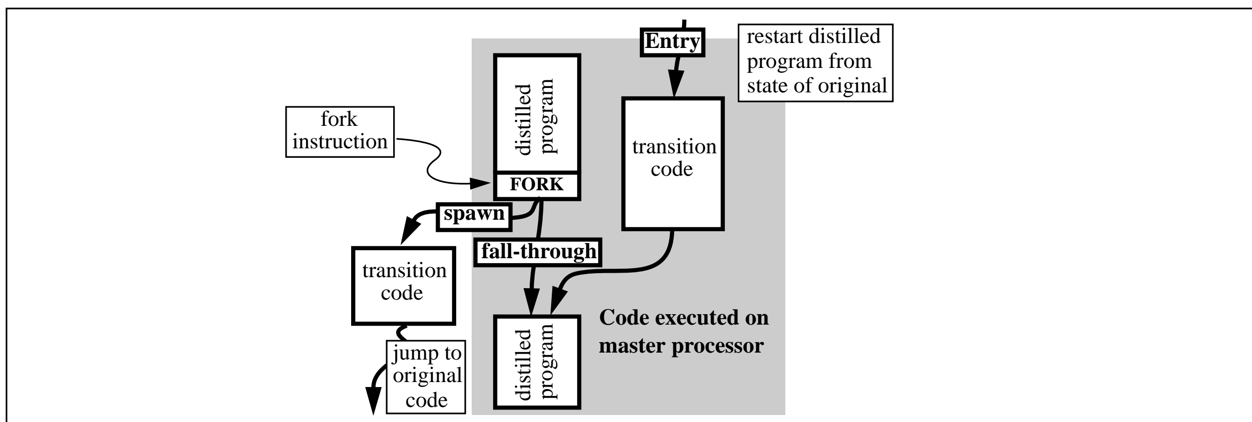

Figure 8. Distilled program structure to support checkpointing and misspeculation recovery. Basic blocks ending in **FORK** instructions continue executing the **fall-through** path on the master processor and **spawn** task execution on an idle slave processor. Entries, with associated transition code, enable restarting the master after a misspeculation.

University of Wisconsin-Madison Technical Report 1438. Available from http://www.cs.wisc.edu/~zilles/papers/mssp-tr1438.pdf

3.3 Misspeculation Recovery

Because the distilled program is optimized for the common case, it will cause a misspeculation when an uncommon case is encountered. As discussed in Section 2.3, these misspeculations will be detected by comparing the live-in state (including the starting task PC) with the architected state resulting from the previous task. When a misspeculation is detected, all checkpoints and speculative state are destroyed; the architected state is as it was left by the last task to retire.

To restart the distilled program after a misspeculation, we provide an *entry* associated with each checkpoint. This entry contains transition code—in the same spirit as for the spawn—that converts the state of the original program into that of the distilled program before jumping into the distilled program. Finding the entry PC in the distilled program corresponding to a location in the original program is discussed in the next section.

3.4 Mapping Between Original and Distilled Programs

In this proposed implementation, the distilled program static code image is distinct from the original program. While using a separate code image provides flexibility in transforming the original program into the distilled program, it requires us to provide explicit mappings between the two programs, because the distilled program resides at a different set of memory addresses than the original program. In this section, we discuss the two situations for which these mappings must be used: (1) instances in the (original) program when program counters (PCs) are stored in registers and memory, and (2) transitions from one program to the other.

Although each program has a separate code image, they share a single data memory image (*e.g.*, stack, global segment, etc.). All PCs in this image must point to the original program, as they would in a traditional execution. Thus, unless we perform some translation, any indirect branch (*e.g.,* a virtual function call or return) performed by the distilled program will thrust it into the original program's code segment. To avoid this transition, all indirect branches first translate the target PC using a map from the original program to the speculative program[4]. The map need only include PCs in the original program that are targets of indirect branches.

Similarly, whenever the distilled program generates a PC (*e.g.,* a return address from a jump-and-link instruction) that may be part of a checkpoint, the generated address must be the PC that would be generated by the original program. Otherwise, any use of that value in the original program will result in a verification failure. Thus, return addresses must be translated, but hardware translation is not necessary

3. The first instruction in a task will have it's checkpoint bit set; obviously this stop bit is ignored.
4. Our model does the translation in hardware as part of the indirect branch's execution, but other implementations are possible.

University of Wisconsin-Madison Technical Report 1438. Available from http://www.cs.wisc.edu/~zilles/papers/mssp-tr1438.pdf

because the return address is a constant. Instead, we can statically translate it, build it as a constant in the distilled program, and use a non-linking jump instruction.[5]

The above mappings are necessary to avoid undesired inter-program transitions, but we need to provide additional mappings when such transitions are desired. As part of a checkpoint, the master must include the starting PC for the task. This, like the return address translation, discussed above, can be statically translated. In addition, the misspeculation recovery process involves restarting the master at the entry point that corresponds to the original program's current location. This again involves a translation, from original to distilled program PCs, that is performed by the recovery hardware. There is one entry corresponding to each checkpoint location in the original program, since the distilled program is always restarted at a checkpoint boundary.

Both mappings from the original program to the distilled program (*i.e.,* entries and indirect branches) must be one-to-one mappings[6]. This constraint prevents replicating regions that require these mappings, disallowing some instances of optimizations like function inlining or code specialization.

4 Analytical Performance Model

From the execution model described in Section 3 we now present a simple analytical model to allow us to reason about performance. Our model makes the following assumptions:

- All tasks are equivalent and have execution time, **E**.
- Distilling the program results in a speedup of α; distilled program segments execute in $\mathbf{E}/\alpha$ time.
- There is an initiation latency, **I**, between when a fork instruction is executed by the distilled program and when the task begins. This latency accounts for inter-core communication latency, time to execute transition code, and any additional execution latency incurred due to branch mispredictions or cache misses not observed by a sequential execution.
- There is a binomial distribution with some probability, **P**, that a checkpoint received by a task will be correctly verified.[7]
- Misspeculations are detected with a latency, **D**, after the previous task has been completed. This latency accounts for the time to update architected state and the inter-core communication required to check the misspeculated task's live-ins.
- Restarting the distilled program takes a latency, **R**, after a misspeculation has been detected. This latency accounts for any inter-core communication to transfer architected state and for the time required to execute transition code.
- Additional slave processors are always available. Thus, verification is on the critical path only for tasks that correspond to distilled program segments that produce incorrect checkpoints.

5. Although this non-linking jump does not write a register we still want it to push the return address on the return address stack (RAS). The master processor could interpret the existing JAL instructions to have this behavior.
6. A mechanism could be provided to select between mappings to remove this constraint.
7. The correctness of checkpoints is assumed to be independent and identically distributed (IID).

University of Wisconsin-Madison Technical Report 1438. Available from http://www.cs.wisc.edu/~zilles/papers/mssp-tr1438.pdf

The original execution time for a program composed of **N** tasks is **NE**. The execution time of each task in the MSSP execution depends on whether its segment in the distilled program produced a correct checkpoint. If so, the task's execution time is that of the distilled program's segment, **E/α**. If not, the task's execution time is it's latency, **E**, plus the initiation, detection, and restart latencies, **I+D+R**. (For algebraic simplicity, we group these terms into a single normalized overhead term, **O = (I+D+R)/E**). The frequency of these events are **P** and **(1 - P)**, respectively. Thus, the total execution time is **N(pE/α + (1-p)E(1+O))**, and speedup is given by:

$$speedup = \frac{time(sequential)}{time(parallel)} - 1 = \left(\frac{NE}{N\left(\frac{PE}{\alpha} + (1-P)E(1+O)\right)} \right) - 1 = \frac{1}{\frac{P}{\alpha} + (1-P)\langle 1+O \rangle} - 1$$

The resulting equation has three free variables: α, **P**, and **O**. Figure 9(a) shows that if we assume the normalized overhead, **O**, is 1 (*i.e.,* equal to the task execution time), then ***speedup is super-linear with prediction accuracy***. As is expected, at low prediction accuracies slow-downs are incurred. At high accuracies (*i.e.,* **P** > .98), ***performance closely tracks the performance of the distilled program***. Sensitivity to normalized overhead is shown in Figure 9(b). This plot demonstrates that ***the architecture is largely insensitive to inter-core latency*** when prediction accuracy is high. The parameters α, and **P** are properties of the distilled program. We explore the interaction between these terms in the next section.

5 Initial Exploration of Distilled Programs

In this section, we describe an initial exploration of distilled programs. This is not intended to be a complete characterization but, instead, is a demonstration that significant potential exists and that the concept warrants further study. This exploration has three components: first, we present a code snippet that was distilled by hand, discussing the optimizations performed and their effectiveness. Second, we present some

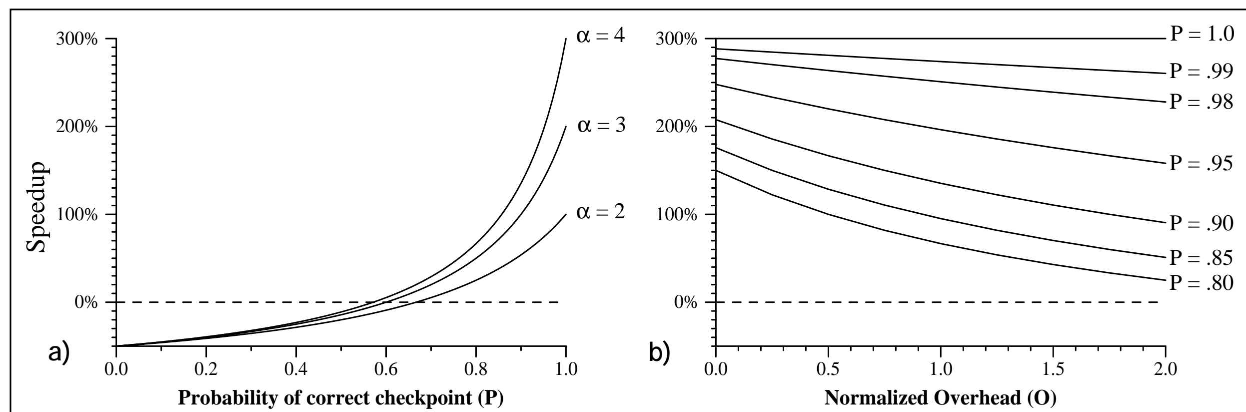

Figure 9. Performance predicted by the analytical model. (a) Speedup is super-linear with checkpoint prediction accuracy, and at high prediction accuracy performance tracks that of the distilled program, (results shown for O=1). (b) The architecture is insensitive to inter-processor communication latency (captured by parameter O) when checkpoint prediction accuracy is high (results shown for α = 4).

University of Wisconsin-Madison Technical Report 1438. Available from http://www.cs.wisc.edu/~zilles/papers/mssp-tr1438.pdf

evidence that suggests that our example is representative of the SPEC 2000 integer benchmarks. Third, we describe our automated distiller and present results for the optimizations implemented. All of our experiments are done in the context of optimized (-O4 -arch ev6 -fast) Alpha ISA binaries using simulators derived from the SimpleScalar toolkit [4].

5.1 Examples of Speculative Transformations

In Section 4, we showed that our proposed execution model is likely to achieve large speedups only if it takes significantly less time to execute the distilled program than the original program and few misspeculations occur. Execution time can be compute as:

$$\textit{execution time} = \frac{\textit{number of dynamic instruction}}{\textit{instructions per cycle (IPC)} \times \textit{frequency}}$$

Since the master processor will have no frequency advantage over the slave processors, its performance advantage must come from reducing the dynamic instruction count and/or increasing the IPC. In Section 5.1.1, we show a code example that can be distilled to 33% of its original dynamic length. In Section 5.1.2, we argue why this code can have an IPC as good or better than the original code.

5.1.1 Reducing Dynamic Instruction Count

In Figure 10(a), we show the control flow graph (CFG) for a pair of functions (`bsR` and `spec_getc`) from the benchmark `bzip2`. `BsR` and its calls to `spec_getc` comprise almost 3% of the instructions executed in our runs of `bzip2`. Many of the branches are strongly biased or always taken in one direction, resulting in two dominant paths through this code. These paths have dynamic instruction lengths of 34 and 102 instructions.

By applying profile-driven speculative transformations to the code, it can be reduced to the CFG shown in Figure 10(b). The two dominant paths have been reduced to 15 and 30 instructions, respectively. All other paths have been (speculatively) optimized away. On the infrequent executions of these paths—84 times in roughly 10 million executions, or about 0.001% of the time—a misspeculation will occur. Otherwise, the optimized code faithfully reproduces the execution behavior of the original code.

In Figure 10(c), we attribute the removal of each instruction to an optimization that enabled it. This classification is not canonical because the elimination of an instruction often requires multiple optimizations, but it provides some insight into the effectiveness of various optimizations. The fruitful optimizations (applied by hand) in this example are:

- **Nop Elimination:** We remove compiler inserted nops. Not an optimization *per se*.
- **Dead Code Elimination:** We remove instructions whose results never affect an active path.
- **Identity Operation Elimination:** The result of some operations is consistently equal to one of its input operands. Most commonly this occurs with logical operations where one operand is always a superset of the other. These instructions can be eliminated.

University of Wisconsin-Madison Technical Report 1438. Available from http://www.cs.wisc.edu/~zilles/papers/mssp-tr1438.pdf

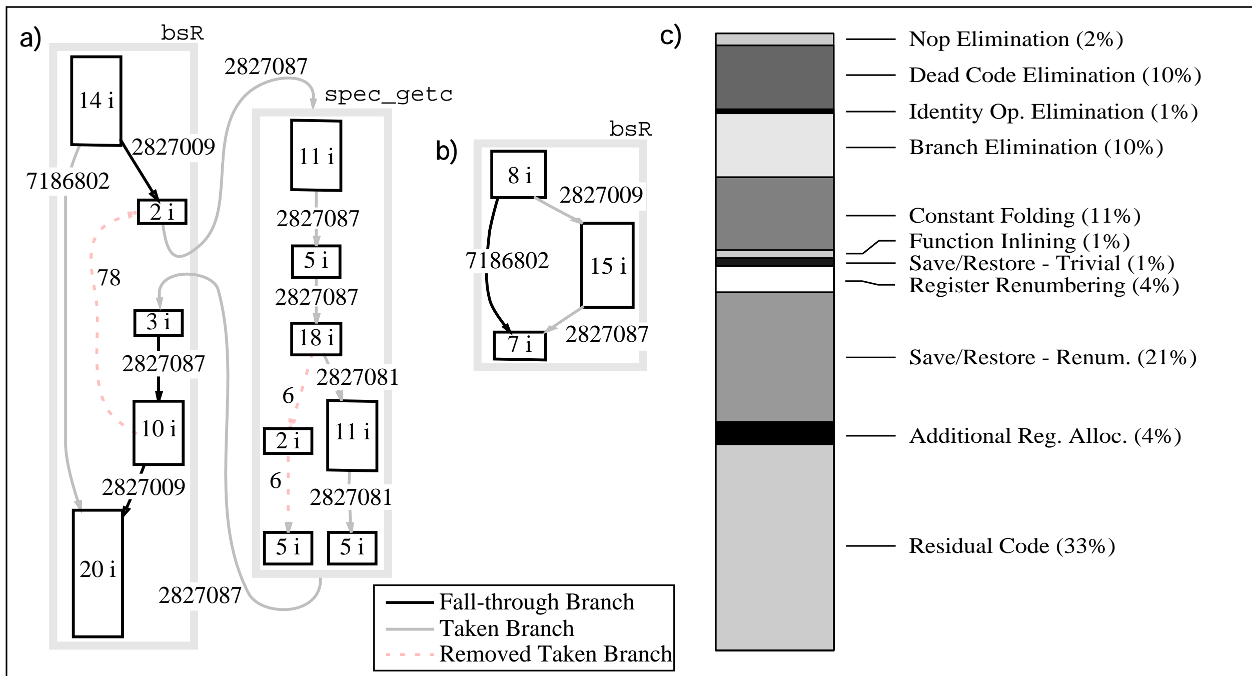

Figure 10. Distilling Programs with Speculative Transformations. A code example (a) can be distilled (b) to reduce average dynamic instruction path length by 67%. (c) We quantify the benefit of each speculative transformation (described in the text). Rounding error causes percentages to not sum to 100%.

- **Branch Elimination:** Strongly-biased branches and their predicate computation can be removed.
- **Constant Folding:** Constants can be pushed into the offset field of a memory instruction. Stack pointer arithmetic can be collapsed if a function does not call other functions dynamically.
- **Function Inlining:** Inlining directly enables the removal of call and return instructions. More importantly, it allows functions to be specialized to their call site.
- **Save/Restore Removal - Trivial:** Register saves and restores can be removed if the instructions that were using the saved register are eliminated.
- **Register Renumbering:** Many register moves can be eliminated by reassigning architectural register numbers.
- **Save/Restore Removal - Renumbering:** If free registers are available, architectural register numbers can be reassigned to alleviate the need to save and restore registers.
- **Additional Register Allocation:** Repeatedly accessed memory values can be register allocated if free registers are available. Frequently, the compiler is prevented from allocating registers because it cannot prove freedom from aliases. With a memory dependence profile, the distilled program can be attentive to frequent aliases when allocating registers.

These optimizations enable the average dynamic path length through the example to be reduced by 67%. One important observation is that the benefit is achieved through the cooperation of many optimizations. Although the relative contribution of the individual optimizations and the total reduction in path length vary for different pieces of code, it has been our experience that substantial improvements always require the composition of multiple optimizations.

University of Wisconsin-Madison Technical Report 1438. Available from http://www.cs.wisc.edu/~zilles/papers/mssp-tr1438.pdf

5.1.2 Improving IPC

In addition to reducing dynamic instruction count, we believe that the distilled program can have a higher IPC than the original program. Below we describe some ways that this can be achieved. A number of these correspond to traditional profile-directed optimizations. Distilled programs and MSSP provide a vehicle for performing these optimizations at run-time (when the profile information is most relevant) without fear of breaking fragile code. Clearly, the benefit of these techniques will be reduced if the original program has already incorporated them.

- **Speculative Optimizations:** Some of the optimizations described above (*e.g.*, register allocation) not only remove code, but simplify the remaining code. Distilling the example reduces the fraction of loads to 1/5 from 1/4, reducing dataflow height and contention for the cache ports.
- **Scheduling:** Having removed branches, the distilled program has larger basic blocks, which facilitates instruction scheduling. In addition, loads can be hoisted across basic blocks with impunity; exceptions caused by the distilled program are ignored.
- **Reducing Static Code Size:** Removing instructions from active blocks and eliminating inactive blocks reduces static code size, enabling more efficient use of the instruction cache.
- **Code Layout:** Distilling the above example, reduces the average number of discontinuous fetches (*i.e.*, taken branches) by a factor of 4, through function inlining, branch removal, and assigning the dominant branch target to the fall-through path. Code layout can also minimize I-cache conflicts.
- **If-conversion:** Some frequently mispredicted branches can be if-converted using `cmov` instructions to avoid branch misprediction penalties. Distilling programs may create additional profitable opportunities for if-conversion by reducing the code in the if and else clauses.
- **Pre-fetching:** Cache miss profiling can guide scheduling of loads and insertion of pre-fetches.

We expect these optimizations to maintain, if not improve, IPC relative to the original program.

5.2 Predictability and Repetition in Programs

In the previous section, we demonstrated a example code segment that could be distilled to a third of its original size with minimal impact on correctness. In this section, we present data on the ubiquity of predictability in non-numeric programs to suggest that the example is representative. We present data on the distribution of branch biases, the presence of code expansion-free inlining opportunities, and the lifetimes of register and memory values.

As demonstrated in Section 5.1.1, much of the computation performed by non-numeric programs to resolve control-flow is unnecessary. Many static branches are only taken in one direction, and these static branches comprise a significant fraction of dynamic branches (*e.g.*, as high as 80% in `vortex`). These branches and others that only rarely go in the non-bias direction can be removed from the distilled program with minimal impact on correctness. Figure 11(a) shows that branches with greater than 99% bias (denoted by the dotted vertical line) make up 93%, 72%, 40%, and 28% of the dynamic branches in `vortex`, `gcc`,

University of Wisconsin-Madison Technical Report 1438. Available from http://www.cs.wisc.edu/~zilles/papers/mssp-tr1438.pdf

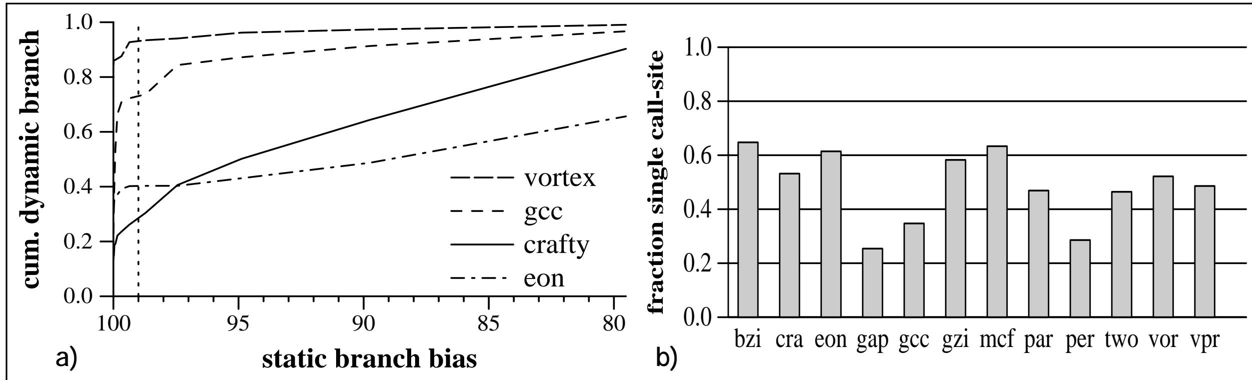

Figure 11. Exploitable regularity in control flow. (a) Highly-biased static branches contribute a large fraction of dynamic branches. Cumulative distributions of dynamic branches—categorized by the static branch's bias—is shown for 4 benchmarks that represent the distribution of behaviors. (b) About one-half of executed functions are called from a single call site and can be inlined with no code growth.

`eon`, and `crafty`. Furthermore, dynamically, many functions have a unique call site. Figure 11(b) shows that about half of the functions touched during the execution can be inlined without code growth.

Most values computed by the program fall into one of the two optimization cases shown in Figure 5: short lifetimes and distant first uses. Almost all register values and 30-40% of stores are not referenced more than 100 instructions after the value is created. Many of these values will be created and killed within a task and therefore need not be included in checkpoints. Another 20-40% of store values are not referenced during the first 10,000 instructions after the store. These stores need not be executed by the distilled program because, by the time the first use is encountered, the value will have been already produced by the original program. Figure 12 shows data for `perl`, which is representative of SpecInt 2000.

5.3 Automatic Program Distillation

As part of our research on the MSSP execution model, we are developing an automatic program distiller. This infrastructure is not complete, but for the optimizations currently implemented we obtain results that

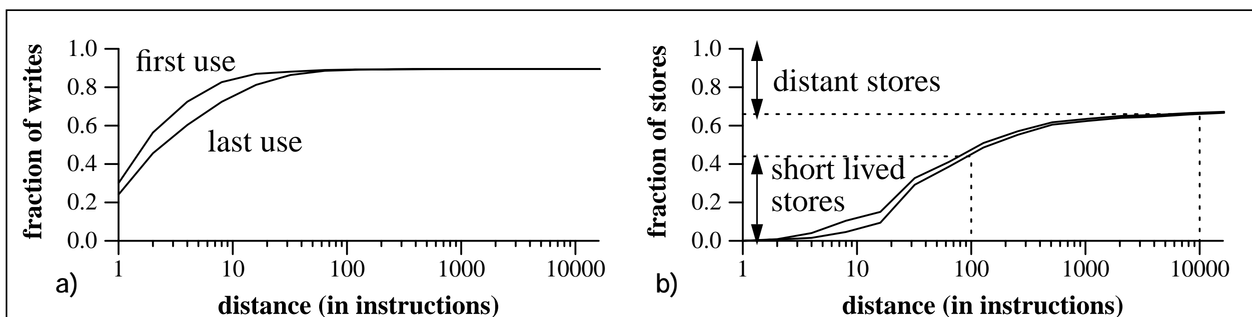

Figure 12. Distances, in instructions, to first and last use from register (a) and memory (b) writes. Data for `perl` plotted as cumulative fraction (*i.e.*, 75% of registers observe their last use within 10 cycles of the write). For both registers and memory, first and last uses are rarely far apart. Almost all register values are read immediately—within 10 instructions—and are never referenced again after 20 instructions. 11% of written registers are never read. Store lifetimes exhibit more variation. The distilled program can potentially avoid 80% of stores: about 45% are short lived (last use within 100 instructions) and another 35% have distant first uses (not referenced in the first 10,000 instructions following the store.)

University of Wisconsin-Madison Technical Report 1438. Available from http://www.cs.wisc.edu/~zilles/papers/mssp-tr1438.pdf

	bzi	cra	eon	gap	gcc	gzi	mcf	par	per	two	vor	vpr
Avg. Task Size	57	121	109	89	58	44	59	93	111	199	98	129
Avg. # live-in reg.	5.9	5.3	5.0	4.8	4.6	4.3	3.2	4.7	4.8	4.6	5.5	6.4
Avg. # live-in mem.	6	24	16	13	7	5	10	10	14	23	12	17

Table 1. Task characterization. Task size is in dynamic instructions. While register live-in count is largely independent of average task size, number of memory live-in values correlates strongly to average task size.

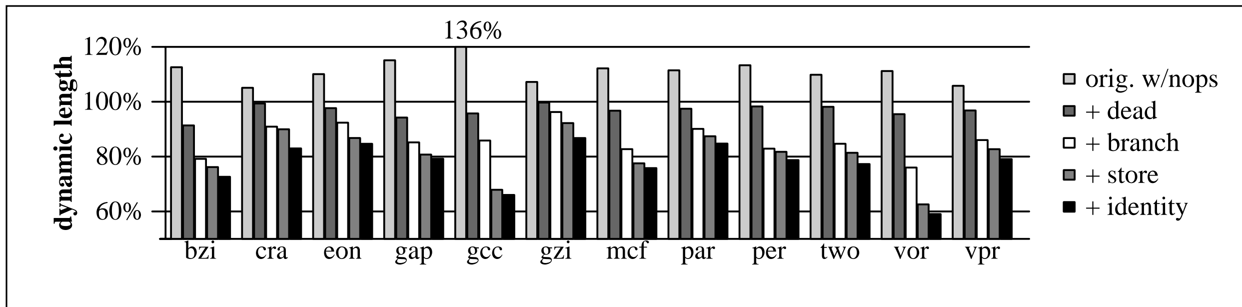

Figure 13. Reduction of dynamic instruction count as a function of the extent of optimization. Optimization groups described in text. Data shown for runs of the first 2 billion instructions of the benchmark, normalized to the original program with the nops removed.

are comparable to the manually distilled example from Section 5.1.1. We have seen no evidence to believe that the automated distiller will fail to achieve the full reduction achieved by the example when all optimizations are completed.

Our infrastructure consists of three pieces: the profiler, the distiller, and the evaluation simulator. The profiler collects a variety of information (*e.g.,* control-flow edge profiles and memory dependence profiles) and saves the data to files for use by the distiller. The distiller generates a internal representation (IR) from the original program. Using heuristics that attempt to create tasks of moderate size (around 100 instructions), checkpoints are inserted[8] into the IR at natural boundaries (*e.g.,* loop headers and return targets) in an attempt to minimize the size of live-in sets. Next the distiller, guided by profile information, applies speculative code transformations to the IR. The code is then generated along with the necessary maps and the checkpoint bitmap. The resulting average task and live-in set sizes for the benchmarks are shown in Table 1.

The evaluation simulator performs functional simulation of both the master and the slaves. The simulator is completely execution-driven and allows arbitrary wrong-path execution. The architected memory, shared by the master and slaves, is not updated immediately by the slaves to simulate the lag between the distilled program execution and task retirement.

For the optimizations implemented, our automatic distiller achieves results comparable to the example presented in Section 5.1.1. Reductions in dynamic instruction counts are shown for a succession of optimizations in Figure 13. The results are normalized to the original program with the nops removed. The num-

8. Some optimizations are sensitive to the location of checkpoints; hence, our results are affected by the quality of our heuristics. We expect that our results could be improved by better checkpoint insertion algorithms.

University of Wisconsin-Madison Technical Report 1438. Available from http://www.cs.wisc.edu/~zilles/papers/mssp-tr1438.pdf

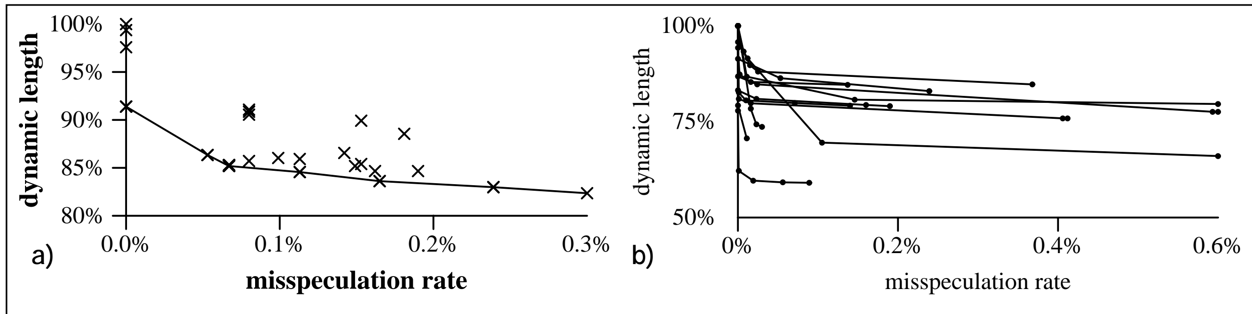

Figure 14. Performance/accuracy trade-off. (a) The misspeculation rates and dynamic path reductions for a spectrum of distilled programs for the benchmark `crafty`, (b) The best configurations for all benchmarks have a similar L-shape trend. Most of the benefit of each optimization comes with only a small misspeculation penalty; as the correctness threshold is lowered only incremental performance is achieved.

ber of nops in the original execution is substantial; the first bar shows the relative size of the **original** program with the nops included. The second bar turns on **dead** code elimination. The third bar adds **branch** removal (using an accuracy threshold of 1%) and inlining achievable without code growth. Bar four adds the distant **store** optimization, which didn't benefit the two functions in the example in Section 5.1.1, but benefits the executions as a whole, especially `gcc` and `vortex`. The fifth and final bar adds the **identity** optimization.

These optimizations reduce dynamic instruction count by 15-40% (20-50% when counting nop removal) across the benchmark suite. These results, for the most part, exceed the 22% reduction the same optimizations achieved in our hand optimized example. All executions had task misspeculation rates below 1% and many configurations were below 0.1%. Thus, it appears that it will be possible to automatically distill programs to achieve significant reductions in dynamic instruction length while maintaining small misspeculation rates.

By varying the enabled set of optimizations and adjusting the correctness thresholds of the individual optimizations we can create a spectrum of distilled programs. Figure 14(a) plots the accuracy of distilled programs for `crafty` against their reduction in dynamic instruction count. The data points above the line are bad configurations (worse accuracy and more dynamic instructions than another configuration). The best configurations for each benchmark follow a trend (shown in Figure 14(b)). Initially, dynamic instruction count falls rapidly with little impact on accuracy. Shortly thereafter, the curves flatten out—at a task misprediction rate of less 0.1%—and additional reduction in dynamic instruction count comes only with increasing the number of misspeculations. The best performance is likely achieved by configurations on the knee of the curve.

6 Related Work

This work draws inspiration from three main bodies of work: speculative multithreading, leader/follower architectures, and speculative compiler optimizations.

University of Wisconsin-Madison Technical Report 1438. Available from http://www.cs.wisc.edu/~zilles/papers/mssp-tr1438.pdf

There has been extensive previous research in speculative multithreading, some examples include [1, 3, 7, 10, 12, 16, 18, 20, 30, 29, 32]. Our management of the speculative state created by the distilled program draws heavily from this research. The idea of predicting task live-ins using traditional value predictors has been previously proposed in [1, 20, 22].

The master/slave architecture of the MSSP model corresponds to the leader/follower architectures proposed for sequential processors. To tolerate memory latency, the decoupled access/execute architecture [26] broke the program into a stream that loaded and stored data values (leader), and a stream that performed non-address computation (follower). Pre-execution proposals [8, 25, 31, 34] execute a speculative subset of the program (leader) to prefetch and generate predictions for the complete execution of the original program (follower).

Leader/follower architectures have also been proposed for fault tolerance, where both processes execute the original program, detecting inconsistencies in the executions. AR-SMT [24] performs the two executions as threads on an SMT. Diva [2] performs the second execution on a simpler processor that can be verified to detect design faults in the core.

Of the leader/follower architectures, the closest to MSSP is SlipStream [31]. The MSSP model differs from SlipStream in three major ways: (1) the follower execution is parallelized in MSSP, (2) the code executed by the leader is a separate static image rather than a strict subset of the original program. A separate image provides additional optimization opportunities, but it requires explicit maps to correlate the two executions. (3) SlipStream dynamically selects the program subset based on the predicted path, where MSSP uses a static distilled program.

Our leader, the distilled program, derives its advantage over the whole execution through applying profile-driven speculative transformations. Others have previously observed the benefit of these transformations, some examples include control-based [6, 9, 11, 19, 27], data-dependence [14, 17, 21], and value-based [5, 13] optimizations. Our work differs from previous work, because the speculation is not verified by the transformed code but by running the original program in parallel.

7 Summary/Conclusion

In this paper, we presented a new execution model, Master/Slave Speculative Parallelization (MSSP), that differs from previous speculative multithreading architectures by way of its master/slave architecture. A key component of this model is the distilled program, a speculative approximation of the original program. By using representative profile information, program distillation appears capable of generating substantially faster code (67% of code was removed from one example) that accurately reproduces the execution behavior of the original code (task misspeculation rates significantly below 1%). We presented

University of Wisconsin-Madison Technical Report 1438. Available from http://www.cs.wisc.edu/~zilles/papers/mssp-tr1438.pdf

an analytical model that suggests that when distilled programs are accurate, the performance of the whole execution can closely track that of the distilled program.

We believe the MSSP execution model conforms to the necessary real world constraints to become widely adopted. Because the original program is used un-modified, there are no necessary 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 our study of distilled programs suggest can be made infrequent.

8 References

[1] H. Akkary and M. A. Driscoll. A Dynamic Multithreading Processor. In *Proceedings of the 31st Annual IEEE/ACM International Symposium on Microarchitecture*, pages 226–236, Nov. 1998.

[2] T. M. Austin. DIVA: A Reliable Substrate for Deep Submicron Microarchitecture Design. In *Proceedings of the 32nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 196–207, Nov. 1999.

[3] D. Bruening, S. Devabhaktuni, and S. Amarasinghe. Softspec: Software-based Speculative Parallelism. In *3rd ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-3)*, Dec. 2000.

[4] D. C. Burger and T. M. 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] P. P. Chang, S. A. Mahlke, W. Y. Chen, N. J. Water, and W. mei W. Hwu. IMPACT: An Architectural Framework for Multiple-Instruction-Issue Processors. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, pages 266–275, May 1991.

[7] M. Cintra, J. Martinez, and J. Torrellas. Architectural Support for Scalable Speculative Parallelization in Shared-Memory Systems. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, June 2000.

[8] J. Collins, H. Wang, D. Tullsen, C. Hughes, Y. Lee, D. Lavery, and J. Shen. Speculative precomputation: Long-range prefetching of delinquent loads. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 14–25, July 2001.

[9] R. P. Colwell, R. P. Nix, J. J. O. Donnell, D. B. Papworth, and P. K. Rodman. A VLIW architecture for a trace scheduling compiler. In *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 180–192, Oct. 1987.

[10] P. Dubey, K. O'Brien, K. O'Brien, and C. Barton. Single-Program Speculative Multithreading (SPSM) Architecture: Compiler-Assisted FineGrained Multithreading. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, June 1995.

[11] J. A. Fisher. Trace scheduling: a technique for global microcode compaction. C-30(7):478–490, 1981.

[12] M. Franklin and G. S. Sohi. The Expandable Split Window Paradigm for Exploiting Fine-Grain Parallelism. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 58–67, May 1992.

[13] C. Fu, M. D. Jennings, S. Y. Larin, and T. M. Conte. Value speculation scheduling for high performance processors. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 262–271, Oct. 1998.

[14] D. M. Gallagher, W. Y. Chen, S. A. Mahlke, J. C. Gyllenhaal, and W. W. Hwu. Dynamic memory disambiguation using the memory conflict buffer. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 183–193, Oct. 1994.

University of Wisconsin-Madison Technical Report 1438. Available from http://www.cs.wisc.edu/~zilles/papers/mssp-tr1438.pdf

[15] S. Gopal, T. Vijaykumar, J. E. Smith, and G. S. Sohi. Speculative Versioning Cache. In *Proceedings of the Fourth IEEE Symposium on High-Performance Computer Architecture*, pages 195–205, Feb. 1998.

[16] L. Hammond, M. Willey, and K. Olukotun. Data Speculation Support for a Chip Multiprocessor. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 58–69, Oct. 1998.

[17] V. Kathail, M. Schlansker, and B. R. Rau. HPL PlayDoh Architecture Specification: Version 1.0. Technical Report HPL-93-80, HP Laboratories, Feb 1994.

[18] T. Knight. An Architecture for Mostly Functional Languages. In *Proceedings of the ACM Conference on Lisp and Functional Programming*, pages 88–93, Aug. 1986.

[19] S. A. Mahlke, W. Y. Chen, R. A. Bringmann, R. E. Hank, W. W. Hwu, B. R. Rau, and M. S. Schlansker. Sentinel scheduling: A model for compiler-controlled speculative execution. *ACM Transactions on Computer Systems*, 11(4), Nov. 1993.

[20] P. Marcuello, J. Tubella, and A. Gonzalez. Value Prediction for Speculative Multithreaded Architectures. In *Proceedings of the 32nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 219–229, Nov. 1999.

[21] M. Mock, M. Das, C. Chambers, and S. J. Eggers. Dynamic Points-To Sets: A Comparison with Static Analyses and Potential Applications in Program Understanding and Optimization. In *Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, June 2001.

[22] J. Oplinger, D. Heine, and M. S. Lam. In Search of Speculative Thread-Level Parallelism. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, Oct. 1999.

[23] R. Rajwar and J. R. Goodman. Speculative Lock Elision: Enabling Highly Concurrent Multithreaded Execution. In *Proceedings of the 34rd Annual IEEE/ACM International Symposium on Microarchitecture*, Dec. 2001.

[24] E. Rotenberg. AR-SMT: A Microarchitectural Approach to Fault Tolerance in Microprocessors. In *Proceedings of the 29th International Symposium on Fault-Tolerant Computing Systems*, pages 84–91, June 1999.

[25] A. Roth and G. Sohi. Speculative Data-Driven Multi-Threading. In *Proceedings of the Seventh IEEE Symposium on High-Performance Computer Architecture*, pages 37–48, Jan. 2001.

[26] J. E. Smith. Decoupled Access/Execute Computer Architecture. In *Proceedings of the 9th Annual Symposium on Computer Architecture*, pages 112–119, Apr. 1982.

[27] M. D. Smith, M. S. Lam, and M. A. Horowitz. Boosting beyond static scheduling in a superscalar processor. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 344–354, May 1990.

[28] A. Sodani and G. S. Sohi. Dynamic Instruction Reuse. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 194–205, June 1997.

[29] G. Sohi, S. Breach, and T. Vijaykumar. Multiscalar Processors. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 414–425, June 1995.

[30] J. G. Steffan and T. C. Mowry. The Potential for Using Thread-Level Data Speculation to Facilitate Automatic Parallelization. In *Proceedings of the Fourth IEEE Symposium on High-Performance Computer Architecture*, Feb. 1998.

[31] K. Sundaramoorthy, Z. Purser, and E. Rotenberg. Slipstream Processors: Improving both Performance and Fault Tolerance. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, Nov. 2000.

[32] J.-Y. Tsai and P.-C. Yew. The Superthreaded Architecture: Thread Pipelining with Run-Time Data Dependence Checking and Control Speculation. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pages 35–46, Oct. 1996.

[33] T. N. Vijaykumar and G. S. Sohi. Task Selection for a Multiscalar Processor. In *Proceedings of the 31st Annual IEEE/ACM International Symposium on Microarchitecture*, pages 81–92, Nov. 1998.

[34] C. B. Zilles and G. S. Sohi. Execution-based Prediction Using Speculative Slices. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 2–13, July 2001.

University of Wisconsin-Madison Technical Report 1438. Available from http://www.cs.wisc.edu/~zilles/papers/mssp-tr1438.pdf