Formally Defining and Verifying Master/Slave Speculative Parallelization*

Pierre Salverda, Grigore Roşu and Craig Zilles

University of Illinois at Urbana-Champaign
{salverda,grosu,zilles}@cs.uiuc.edu

Abstract. Master/Slave Speculative Parallelization (MSSP) is a new execution paradigm that decouples the issues of performance and correctness in microprocessor design and implementation. MSSP uses a fast, not necessarily correct, master processor to speculatively split a program into tasks, which are executed independently and concurrently on slower, but correct, slave processors. This work reports on the first steps in our efforts to formally validate that overall correctness can be achieved in MSSP despite a lack of correctness guarantees in its performance-critical parts. We describe three levels of an abstract model for MSSP, each refining the next and each preserving equivalence to a sequential machine. Equivalence is established in terms of a jumping refinement, a notion we introduce to describe equivalence at specific places of interest in the code. We also report on experiences and insights gained from this exercise. In particular, we show how formalizing MSSP facilitated a deeper understanding of performance-correctness decoupling and its attendant tradeoffs, all key features of the MSSP paradigm. Moreover, formalization revealed all assumptions underpinning correctness, which, being specified abstractly, can be understood in an implementation-independent way. We found these results so valuable that we plan to advance MSSP's formalization in parallel with its subsequent design iterations.

1 Introduction

Technology advances have reached the point where it is now possible to engineer multiple processor cores onto a single chip. While this capability lends itself to throughput-oriented workloads, latency-critical sequential applications do not see any benefit. The traditional approach of relying on the programmer to find and extract parallelism in such programs has met with little success, primarily because manual parallelization is complicated and error-prone. Thus, a fundamental challenge facing computer architects today is bringing the benefits of multiple cores to bear on the performance of sequential programs.

Master/Slave Speculative Parallelization (MSSP) [14] is a recent proposal for automatically extracting parallelism from sequential programs. The paradigm uses a *master* processor to divide the dynamic instruction stream into pieces,

* This work was supported in part by a grant from Intel and National Science Foundation grants CCR-0311340 and CCF-0347260.

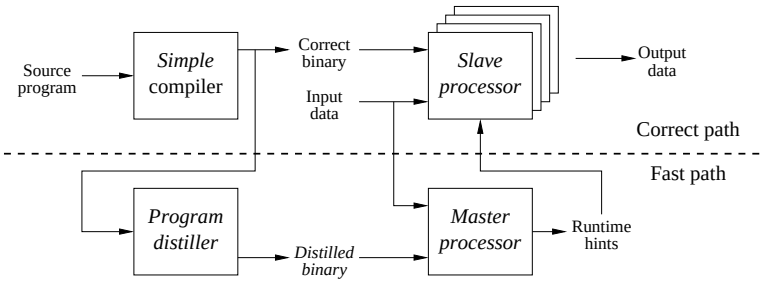

Fig. 1. Conceptual organization of MSSP. *On the fast path, the master executes an approximate program (distilled binary) to run ahead of the slaves, providing hints (live-ins) of where execution is likely to be headed. On the slower correct path, the slave processors use the hints to concurrently compute their tasks.*

called *tasks*, which are executed in parallel on multiple *slave* processors. Dataflow dependences between the tasks are resolved by the master, which predicts *live-in* values for each task by executing an *approximate* version of the original program. Slaves use the original program code when executing their tasks, but operate on the speculative live-in data supplied to them by the master. The results computed by slaves are committed to the machine's non-speculative state only if the corresponding live-ins are consistent with that state. If inconsistencies are detected, the results are discarded and the machine resumes its operation using the pristine non-speculative state as a starting point. Because slaves operate concurrently, overall performance is determined largely by the master processor. In turn, because the master executes an approximate (shorter) version of the original program, MSSP is able to achieve significant speed-ups over speculative, out-of-order superscalar machines [14].

The potential for high performance in MSSP is underpinned by its ability to *decouple* performance and correctness concerns, in so doing facilitating the simultaneous pursuit of these otherwise conflicting goals. *Complexity* lies at the root of the tension between performance and correctness: pursuit of the former incurs complexity (out-of-order superscalar architectures and optimizing code transformations are complex), which, in turn, compromises our ability to ensure the latter (complex systems are hard to verify). MSSP decouples the two by separating the parts of the system that produce output — and are thereby constrained to be correct — from the parts that determine the rate at which output is produced. Figure 1 depicts this idea.

Decoupling in such a framework is successful if neither subsystem can compromise the objectives of the other. That is, the fast path should not compromise correctness and the correct path should not determine overall performance. As noted already, the latter property does indeed hold in MSSP because the master processor resides on the critical path, not the slaves. The former requirement — that of correctness in spite of the potential for errors on the fast path — is the focus of this paper.

Thus, our primary goal here is to demonstrate that correctness in MSSP cannot be influenced by how the master operates, nor by the instructions contained in the distilled binary it executes. In so doing, we conclusively demonstrate that the correct path is properly decoupled from the fast path. This provides a formal basis for a central theme in our work: correctness need not be compromised by the pursuit of performance.

The formalization of MSSP also served a secondary goal of obtaining an abstract model for the new execution paradigm. In developing that model, we have exposed the fundamental aspects of MSSP that underpin its correctness. Specifically, we isolated the notion of *task safety* (Section 4) as the principal condition upon which correct operation rests. That we could distill the correctness requirement so precisely and succinctly was simultaneously surprising and encouraging. Indeed, the mere process of formalizing MSSP has been fundamental to our gaining a deeper understanding of an execution paradigm we previously understood only "intuitively." The lack of implementation-specific detail in the abstract model will also facilitate reasoning about correctness in subsequent iterations of the MSSP design, each of which is likely to be encumbered by the artifacts of technology-driven design trade-offs. These benefits accrue because we embarked on the formal study early on in our research, in contrast to much formal work, which tends to be a "post-mortem" exercise whose sole goal is to find errors in an extant design. In this respect, our experiences are in agreement with previous assertions (see, for example, [5]) that formal verification should be a part of the design process.

In terms of the formalization itself, we report on our use of rewriting logic [7], as supported by Maude [3]. We establish correctness by proving MSSP's equivalence to a conventional sequential execution model. In this respect, our work is similar to the extensive studies of microarchitecture verification, where correctness is proven by comparing a microarchitectural specification to the specification of an instruction set architecture (ISA). The work of Burch and Dill [2], Hunt and Sawada [10, 11] and Arvind and Shen [1] are notable examples in this area. However, we differ from those studies in that we are *not* trying to establish that a refinement — in the usual sense of the word — of an ISA is correct, since MSSP is *not* a standard refinement of an ISA.

Although an MSSP machine implements a conventional sequential ISA, it differs from a sequential machine in terms of the granularity at which updates the architected state[1] occur — MSSP updates state at task boundaries rather than at instruction boundaries. Thus, a key property of MSSP is that it "jumps" over sequences of states in the sequential model. In fact, if one ignores MSSP transitions that do not change architected state, then the sequential model can be regarded as a stuttering refinement [6] of MSSP. But this is somewhat counterintuitive because our objective is to reason about MSSP in terms of the sequential model, not the other way round. To capture the desired relationship between

[1] By this we mean the ISA-visible state — the set of all registers and memory cells accessible via the instruction set. Internal state, such as that held speculatively by the master and all slave processors, is not included in this set.

MSSP and the sequential model, we define the notion of *jumping refinement* in Section 3 and then show formally that MSSP is a jumping refinement of the sequential model.

We tackled the formalization process iteratively, beginning with a high-level abstract model in which we make a number of simplifying assumptions. Section 4 describes this work. Sections 5 and 6 show successive refinements of the abstract model, identifying low-level requirements from which our initial assumptions can be inferred. Throughout, we present only the most important and interesting results at an abstract, mathematical level, rather than our particular formalization in Maude. Even though Maude provides a suite of useful tools for our project, we would like to avoid giving the reader the impression that it was a crucial part of our formalization. We believe that one can relatively easily adapt our work to other formal systems and tools. Section 7 concludes the paper and summarizes some of our main observations and lessons learned during this work.

2 An overview of MSSP

In this section, we present an overview of MSSP. This high-level description is meant to provide the contextual knowledge necessary to understand the formal work that follows in the remainder of the paper. A more extensive treatment of MSSP can be found in [13] and [14].

2.1 High-level operation

Consider again Figure 1. An MSSP machine has two execution paths: the fast path and the correct path. The fast path is composed of a single, complex *master* processor that executes a speculatively optimized executable called the *distilled program*. The master processor runs ahead of the correct path execution to produce hints of where the execution is headed. The correct path is implemented by multiple *slave* processors, which lag behind the master. Because the individual slave processors are slower than the master, we need a means for the correct path to keep up. MSSP uses *speculative parallelization* [12] for this purpose. Execution of the correct path program is split into segments, called *tasks*, that are executed concurrently on the slaves.

To enable these tasks to execute independently and in parallel, the master execution is used to predict the sequence of tasks — that is, the starting program counter (PC) of each task, and the values that are live-in to each of them. The predictions are generated by logically taking a checkpoint of the master's (speculative) state at the point corresponding to the beginning of the task.

Because the master's predictions are not guaranteed to be correct, the results computed by slaves are themselves speculative, and must be checked before they can be made architecturally visible. To enable this, each task's inputs (live-ins) and outputs (live-outs) are recorded and sent to a *verification/commit unit*. When a completed task becomes the oldest (*i.e.*, the next to commit), a memoization-like operation is performed that commits the outputs if the inputs match the machine's current architected state.

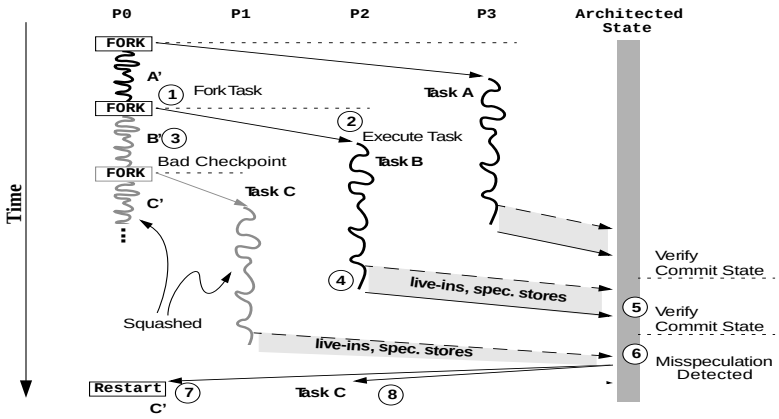

Fig. 2. Master processor distributes checkpoints to slaves. *The master, executing the distilled program on processor P0, assigns tasks to slave processors, providing them with predicted live-in values.*

2.2 MSSP example

To facilitate a conceptual understanding of MSSP, we provide an example that outlines its basic behavior. Figure 2 illustrates an MSSP execution with four processors: one master (P0) and three slaves (P1, P2, and P3) that begin the example idle. Each processor has its own register file and local first-level cache; values held there are speculative. The machine's architected state appears on the right in the figure. This holds the current (correct) values of all ISA-visible registers and memory addresses. In an MSSP machine, this is maintained in the shared second-level cache, which is backed by DRAM.

At annotation (1) in the figure, the master processor spawns Task B onto processor P2, which then begins executing (2). P0 continues executing (3) the distilled program segment that corresponds to Task B, which we refer to as Task B′. As the slave executes Task B, it reads values that it did not write (the live-in values supplied by the master) and performs writes of its own (the live-out values). When Task B completes (4), P2 sends its live-in and live-out values to a verify/commit unit, which checks that the live-ins exactly correspond to the architected state; if so, the live-outs can be committed to architected state (5). The commit is implemented so that it appears atomic to all processors in the system [14]. This avoids potential problems with memory coherence if MSSP is used in a multiprocessor system.

If the master generates an incorrect value (3), one of the recorded live-in values will differ from the corresponding value in the architected state, and a mismatch will be detected at verification (6). When this occurs, the master and all other in-flight tasks are *squashed* — the speculative data they hold in their registers and caches is discarded. The architected state of the machine is not affected by the misspeculation, so it holds the state the program was in at the completion of Task B. At this time, the master is restarted at C′ (7) and,

in parallel, non-speculative execution of the corresponding task in the original program (Task C) begins on P2 (8). In both cases, the processors have their state seeded with the correct values currently held in architected state.

3 Rewriting Logic, Jumping Refinements and Maude

We chose rewriting logic as the formal framework in which to define and reason about MSSP. Rewriting logic (RL) has been introduced as a unifying framework for concurrency [7], making it quite appealing for a complex and highly concurrent architecture like MSSP. As a tool supporting RL, we chose the Maude system [3]. Maude provides a series of formal analysis tools for rewriting logic theories, including: (1) a highly-efficient rewriting engine; (2) a search procedure exploring the (potentially unbounded) state space using a breadth-first strategy; (3) a linear temporal logic (LTL) model checker; and (4) an inductive theorem prover and proof assistant (ITP) [4]. While our present work does not require all of these features, the potential to expand without changing tools makes Maude a compelling choice.

RL extends equational logic by adding rewriting rules as parameterized state transitions. Briefly, a rewriting theory $\mathcal{R}$ is a triple (Σ, E, R), where Σ contains all the type and operator declarations, E contains a set of equations, and R is a set of rewriting rules. Equations in E are used to define the computational infrastructure of a system specification (such as predicates and sets of tasks, in the case of MSSP), while the rewriting rules in R are used to specify the *concurrent* aspects (such as committing a slave processor's live-outs). Equations and rules can contain variables, and they are applied to a given term at any position where they match. Given $\mathcal{R} = (\Sigma, E, R)$, we let $\equiv_{\mathcal{R}}$ and $\Rightarrow_{\mathcal{R}}$ denote the binary relations on terms derived by applying the equations and the rewriting rules, respectively; $\Rightarrow^{\star}_{\mathcal{R}}$ denotes the reflexive, transitive, and $\equiv_{\mathcal{R}}$-closure (*i.e.* modulo equations in E) of $\Rightarrow_{\mathcal{R}}$. The subscript $\mathcal{R}$ is omitted whenever apparent from context.

Any transition system can be defined as a rewrite theory. In particular, both the sequential model and the various versions of MSSP are rewrite theories, each having a special *state* type (or sort); the rewrite sequences on state terms correspond to state transitions. The standard notion of (stuttering) refinement of rewrite theories states that a step in the abstract theory can be simulated by a sequence of steps in the refined theory. We would clearly want MSSP to refine the sequential model, but note that this is not true within the standard meaning of (stuttering) refinement — and this has nothing to do with our choice of using rewriting logic for our formal framework — because MSSP deliberately does not reproduce all the steps of the sequential model. To formally capture this relationship, we introduce the notion of *jumping refinement*, as follows.

Definition 1. *Given $\mathcal{R} = (\Sigma, E, R)$ and $\mathcal{R}' = (\Sigma', E', R')$ with rewrite relations $\Rightarrow_{\mathcal{R}}$ and $\Rightarrow_{\mathcal{R}'}$, respectively, and containing some designated sorts State and State$'$, respectively, together with a map ψ associating terms of sort State to*

terms of sort State′, we say $\mathcal{R}'$ **is a jumping** ψ**-refinement of** $\mathcal{R}$ *iff for any transition* $t \Rightarrow_{\mathcal{R}'} u$ *in* $\mathcal{R}'$ *there is a sequence* $\psi(t) \Rightarrow^{\star}_{\mathcal{R}} \psi(u)$ *of transitions in* $\mathcal{R}$.

The intuition here is that the states in $\mathcal{R}'$ contain more information than those in $\mathcal{R}$, and ψ is a projection extracting a state of $\mathcal{R}$ from a state of $\mathcal{R}'$. It may therefore be the case that several transitions take place in $\mathcal{R}'$ without changing the corresponding state in $\mathcal{R}$. In other words, it may be the case that $t \Rightarrow_{\mathcal{R}'} t'$ while $\psi(t) \equiv_{\mathcal{R}} \psi(t')$, but it is also possible that $\psi(t) \Rightarrow^{+}_{\mathcal{R}} \psi(t')$ for a large number of transitions in $\mathcal{R}$. In the first case we metaphorically say that the transition in $\mathcal{R}'$ "accumulates energy" with respect to $\mathcal{R}$, and in the second that the transition in $\mathcal{R}'$ "jumps" with respect to $\mathcal{R}$. In our case, $\mathcal{R}'$ will specify MSSP, $\mathcal{R}$ the sequential model, and ψ will return the architected state of MSSP. Note that the slave execution steps in MSSP do not modify the architected state, so their execution "accumulates energy"; but once a slave computation is committed, the MSSP machine "jumps" several sequential states.

4 First iteration

We next introduce abstract models for sequential and MSSP execution, and then show that the latter is a jumping refinement of the former. Since the sequential machine model under consideration is deterministic, its executions can safely be considered atomic. This implies the rewrite rules (transitions) in the sequential model can be regarded as equations, so we will often say, by slight abuse of language, that MSSP is *equivalent* to the sequential model instead of a jumping refinement of it.

The formalisms presented here are abstracted from our original *Maude* source, which is harder to read but available online in complete form at [8]. The reader is encouraged to refer to that source for a mechanical formalization, both of the execution model specifications and of the proofs of the main results, the details of which we must necessarily omit here.

4.1 The sequential execution model

The sequential execution model, which we denote by SEQ, serves as a reference against which correctness of MSSP is measured. Since we do not wish to couple ourselves to any particular sequential ISA, we avoid specifying one for SEQ. We can afford to do so because we assume that the slaves implement the same ISA as the "reference" sequential machine.

The SEQ model is centered on the notion of machine state. Although we have defined machine state precisely in other work [9], the abstractions we present in this paper are at a sufficiently high level for us to avoid having to impose a structure on it. Thus, machine state is defined simply as the domain, denoted by $\mathcal{S}$, in which execution occurs. That said, it is useful, in this and subsequent sections, to understand — if only informally — that a member of $\mathcal{S}$ captures the values held in a machine's ISA-visible storage cells (registers and memory

locations). We will see in Section 5 that the live-in and live-out data processed by MSSP slaves also constitute machine states, but that these sets will generally contain members for only a subset of all ISA-visible cells. That is, a machine state need not hold members for *all* ISA-visible cells.

Executing an instruction results in updates to a machine's storage cells, so an instruction's execution constitutes a transformation of machine state. Sequential execution of more instructions is then defined as follows.

Definition 2 (Sequential execution). *Function* $seq : \mathcal{S} \times \mathbb{Z}^+ \mapsto \mathcal{S}$ *models the sequential execution of multiple instructions, and is defined:*

$$seq(S, n) = \begin{cases} S & \text{if } n = 0 \\ seq(next(S), n-1) & \text{otherwise} \end{cases}$$

Function $next : \mathcal{S} \mapsto \mathcal{S}$*, which is uninterpreted, models the execution of a single instruction.*

Note that $seq(S_0, n) = S_1$ states only that S_1 is the state that results after executing n instructions in state S_0. S_0 determines those instructions implicitly, since a machine's state holds both instructions and data. The program counter, itself a member of S_0, identifies the cell in which the next instruction is held.

4.2 The MSSP execution model

The design of a realistic MSSP machine is encumbered by numerous performance-mandated features, none of which have any bearing on the processes that underly its correct operation. Thus, our formalisms are based not on the operation of a real MSSP machine, but on a more abstract model [9] that eliminates all of the performance-related complexities. A few differences between the abstract model and the real machine are worth noting.

First, we view the master as a "black box" that is capable of generating *arbitrary* live-in data. This is of course key to our objective of ensuring that correctness in MSSP is entirely independent of how the fast path operates. This view does, however, expose a limitation of the model: we cannot guarantee forward progress if we cannot guarantee anything about live-in data. But this is an artifact of our model, not of the real MSSP machine, which can make guarantees about forward progress because it has the capability to revert, at any time, to normal sequential execution. In the interests of keeping our formalisms simple, we choose not to model this dual-mode operation in MSSP, and thus treat forward progress as a property that can be verified separate from this work.

Second, rather than have each task's boundaries specified in terms of start and end program counter values, the abstract model assumes tasks are delineated by means of an instruction count — a task is complete when the specified number of instructions have been executed. This simplifies our work because it eliminates the need to expose the notion of program counter in the formal models.

Finally, a real MSSP machine permits slaves to read (but *not* to write) architected state, which, in turn, allows the master to supply as live-in data only

that which has been modified by it; values not modified recently are fetched by a slave direct from architected state.[2] Rather than encumber our model with these details, we assume the master supplies *all* data that it assumes a slave will need; slaves are wholly isolated from architected state (and one another) in our abstract model.

Analogous to the sequential model, MSSP's execution is defined in terms of state transitions, but now manipulation of state occurs at the granularity of *tasks* rather than instructions. MSSP contains a collection of "active" tasks. At each step, the machine selects one task from this collection and, if certain conditions are met (the *task safety* requirement), "commits" it to the architected state. In this section, we do not specify a structure for tasks, nor do we define how the commit process is effected. We state only that if a task satisfies the commit requirement, then committing it has the same effect as advancing the architected state according to the sequential model; we use $\#t$ to denote the number of instructions by which this advancing occurs.

In this section, *task safety* is not interpreted; we define it only as a necessary condition for committing a task, and hence for advancing the architected state. Task safety is a property both of the task to be committed and the machine state to which the commit is to occur. Changes to machine state can thus establish or violate the safety of a given task. Hence, committing one task can affect the safety of another.

We let $\mathcal{T}$ denote the set of all MSSP tasks, $\mathcal{T}^*$ *the set* of all finite sets of tasks, and use operator $| : \mathcal{T}^* \times \mathcal{T}^* \mapsto \mathcal{T}^*$ to construct new task sets; it is both associative and commutative.[3] $|$ is overloaded to also permit construction of task sets from individual tasks.

Definition 3 (MSSP execution). *Function* $mssp : \mathcal{S} \times \mathcal{T}^* \mapsto \mathcal{S}$ *models a single step in the operation of an MSSP machine. For* $t \in \mathcal{T}$, $\tau \in \mathcal{T}^*$ *and* $S \in \mathcal{S}$ *such that* t *is safe for* S, *we define the rule* $mssp(S, t|\tau) \Rightarrow mssp(seq(S, \#t), \tau)$. *To define MSSP operation on the empty task set, we add rule* $mssp(S, \emptyset) \Rightarrow S$.

Note that the above rule is *conditional* — t must be safe for S for the state transition to apply, so at this point MSSP's behavior is left undefined for tasks that are not safe. Note also that the task that is selected for committing is not prescribed, since $|$ is associative and commutative. Indeed, a key property of our model is that *we do not impose an ordering* on the sequence of task commits.

That ordering of commits is not important was initially surprising to us, mainly because the extant MSSP design does impose an order. In fact, we discovered this as a direct result of formalizing MSSP, which thus helped us discover that task safety is the single requirement for correct operation. Further, in eliminating ordering from our model, we impose minimal constraints on implementations while still allowing for reasoning about correctness.

[2] This is merely a performance-driven design choice: the master could equally supply all data, but that would demand too much bandwidth between it and each slave.

[3] We distinguish $|$ from set union ($\cup$) because, in our Maude framework, $\mathcal{T}$ is not a set, but a multiset — it can contain duplicate members.

We point out again that our model for MSSP is devoid of any mention of the master processor. The omission is deliberate — correct operation of the MSSP machine is not dependent on what the master does. Correctness depends only on the slaves and the manner in which the results of their execution are committed.

4.3 Equivalence

We need to show that any transformation of state that can be effected by MSSP can also be achieved in SEQ. In what follows, we first show how we used Maude to arrive at a slightly weaker result — that MSSP *can* effect sequential transformations of machine state — and then a stronger result — that *all* transformations that can be achieved by MSSP are also possible in SEQ.

Equivalence on safe task sets. We can extend Definition 3 to describe MSSP operation at the more coarse granularity of a task set. This requires an extension of our notion of task safety: a task set is considered safe for a given machine state if *there exists* some enumeration of its members such that each is safe for the machine state resulting from committing its predecessor. A simple inductive argument then gives us the following.

Lemma 1. *If $\tau \in \mathcal{T}^*$ is safe for S, then $mssp(S, \tau) \Rightarrow^* mssp(seq(S, \#\tau), \emptyset) \Rightarrow seq(S, \#\tau)$.*

The above lemma states that an MSSP machine starting out in state S and with an active set of tasks τ, which is safe for S, *can* attain the same configuration as a sequential machine executing $\#\tau$ instructions from the same S. This is because safety of a task set is defined in terms of the *existence* of a safe enumeration of its members, yet MSSP operation is not constrained to follow the order of such an enumeration. We therefore cannot infer that MSSP necessarily commits all members of a safe set of tasks; it will do so only if it chooses the right commit order. If it chooses poorly, it can commit some task which, despite being safe, is not the next task in any safe enumeration of τ. In such a case, the remaining members of τ can be rendered unsafe.

It is important to realize that even though our model permits MSSP to pick an inappropriate task, it is never wrong for that task to be committed — since it was safe, committing it, by definition, advances architected state as per SEQ. Choosing an inappropriate task affects only the *efficiency* of the machine, not its correctness, as shown next.

Equivalence for all task sets. We can easily extend the above result to cater for any collection of tasks in the active set. To do so, we define $mssp(S, \tau) = mssp(S, \emptyset)$ for all $\tau \in \mathcal{T}^*$ that contain no tasks safe for S. Thus, if the machine chooses to commit a task that renders the remainder of its active task set unsafe, it simply discards what remains. Hence our earlier claim that inappropriate

choices affect only the machine's efficiency: the order in which tasks are committed determines the fraction of the active set that can be committed before the remainder is discarded.

We are now in a position to define MSSP operation on any given task set. Our main result rests on the claim that any given task set can be partitioned into two disjoint subsets, one that is safe for the current architected state and one that contains *no* safe members.

Theorem 1. *If τ is safe for machine state S, and τ' contains no members that are safe for $seq(S, \#\tau)$, then $mssp(S, \tau|\tau') \Rightarrow^\star mssp(seq(S, \#\tau), \emptyset) \Rightarrow seq(S, \#\tau)$.*

In the above, we assert the existence of a sequence of transitions; we do *not* claim that, given task set $\tau|\tau'$, the MSSP machine will necessarily reach state $seq(S, \#\tau)$. However, we can invoke a "meta-argument" about our specification to prove that any trace of MSSP execution on a given set of tasks effects a bisection of that set into a safe subset and a subset that contains no safe members, so Theorem 1 applies to any execution of the MSSP machine. In other words, all executions in MSSP are possible in SEQ.

5 Second iteration

In the previous section, tasks were uninterpreted and effectively treated as the atomic units on which execution of an MSSP machine is based. Likewise, task safety and the commit operation were uninterpreted and treated as basic capabilities of the machine. In this section, we zoom in on the domain $\mathcal{T}$ by imposing a structure on tasks, which yields a *stuttering refinement* [6] of the MSSP execution model. Thus, we now describe MSSP's execution at an instruction rather than at a task granularity. In so doing, we also partially interpret both task safety and the commit process; Section 6 further refines those concepts.

Once again, we point out that the results presented here are distilled from the original Maude specifications [8], to which we refer the interested reader.

5.1 Tasks

In the existing MSSP implementation, a task is constructed by the master processor, then transferred to a slave where it executes to completion, and finally checked by the verification unit, which either commits or discards the results. Consequently, we define a task as a tuple comprising input and output data, plus information about the current state of the execution at a slave processor.

Definition 4 (Task). *A task is a 4-tuple contained in $\mathcal{T} = \mathcal{S} \times \mathbb{Z}^+ \times \mathcal{S} \times \mathbb{Z}^+$. The tuple $\langle S_{in}, n, S_{out}, k \rangle \in \mathcal{T}$ denotes a task with live-in set S_{in} and live-out set S_{out}. The value n is the number of sequential instructions that constitute complete execution of this task; k is the number of instructions that have been executed by a slave so far ($0 \leq k \leq n$).*

A newly created task has form $\langle S_{in}, n, S_{in}, 0\rangle$; at its completion, it has form $\langle S_{in}, n, S_{out}, n\rangle$. We will relate S_{in} and S_{out} later in this section.

We define a number of functions on $\mathcal{T}$ for the sake of notational convenience. Let $t = \langle S_{in}, n, S_{out}, k\rangle$. Functions $\mathit{live\_in} : \mathcal{T} \mapsto \mathcal{S}$ and $\mathit{live\_out} : \mathcal{T} \mapsto \mathcal{S}$ produce the live-in and live-out sets for a given task. Thus, $\mathit{live\_in}(t) = S_{in}$ and $\mathit{live\_out}(t) = S_{out}$. Function $\# : \mathcal{T} \mapsto \mathbb{Z}^+$, which we introduced in the previous section, yields the second component of a task: $\#t = n$.

5.2 Task evolution

We use *task evolution* as a means for modeling the manner in which a task is processed by a slave processor. It is defined as follows.

Definition 5 (Task evolution). *Let $\langle S_{in}, n, S_{out}, k\rangle \in \mathcal{T}$ be a task in an MSSP machine's active task set. Then the following transition rule applies.*

$$\langle S_{in}, n, S_{out}, k\rangle \Rightarrow \begin{cases} \langle S_{in}, n, \mathit{next}(S_{out}), k+1\rangle & \text{if } k < n \\ \langle S_{in}, n, S_{out}, n\rangle & \text{otherwise} \end{cases}$$

Note that this rule is decoupled from the specification of the MSSP machine itself, so tasks evolve independent of, and concurrent with, the task commit process defined in the previous section. From the above definition it is also clear that slaves execute according to the sequential model. More precisely, the first step in slave execution simply advances the live-outs as per SEQ: $\mathit{live\_out}(t)$, which is initially the same as $\mathit{live\_in}(t)$, is transformed to $\mathit{next}(\mathit{live\_in}(t))$. Extrapolating, we arrive at the following transition rule.

Lemma 2. $\langle S_{in}, n, S_{in}, 0\rangle \Rightarrow^* \langle S_{in}, n, \mathit{seq}(S_{in}, n), n\rangle$.

In fact, we can say something stronger: since we specify no transition rules other than those in Definition 5, the *only* way in which a task can reach completion is through the sequential advancing of its live-in set. That is, if t is a completed task, then $\mathit{live\_out}(t) = \mathit{seq}(\mathit{live\_in}(t), \#t)$.

5.3 Task safety and commit

Having introduced task evolution, we can now partially interpret task safety. To do so, we introduce the notion of superimposition, which models the commit process. Operator $\_ \leftarrow \_ : \mathcal{S} \times \mathcal{S} \mapsto \mathcal{S}$ denotes the superimposition of one machine state onto another. We do not interpret this operation formally, simply because the domain $\mathcal{S}$ itself remains uninterpreted. However, the intuition behind its operation should be clear: $S_0 \leftarrow S_1$ is the machine state that results when S_0 is *overwritten* by S_1.[4]

Definition 6 (Task safety). $t \in \mathcal{T}$ *safe for* S *if* $\mathit{seq}(S, \#t) = S \leftarrow \mathit{live\_out}(t)$.

Since $\mathit{live\_out}(t) = \mathit{seq}(\mathit{live\_in}(t), \#t)$ at the completion of t (Lemma 2), task safety is equivalently characterized by $\mathit{seq}(S, \#t) = S \leftarrow \mathit{seq}(\mathit{live\_in}(t), \#t)$.

[4] Recall that live-in and live-out sets need not represent the state of a *whole* machine. It can therefore be the case that S_0 refers to storage cells not covered by S_1. Those cells will appear, unchanged, in the superimposition.

5.4 MSSP operation, refined

We can now replace Definition 3 with the following.

Definition 7 (MSSP operation, refined). *Let $t \in \mathcal{T}$ and $\tau \in \mathcal{T}^*$. If S is a state for which t is safe, then $mssp(S, t|\tau) \Rightarrow mssp(S \leftarrow live\_out(t), \tau)$.*

We have argued that at the completion of task t, $live\_out(t) = seq(live\_in(t), \#t)$. Since t is safe for S, $S \leftarrow seq(live\_in(t), \#t) = seq(S, \#t)$, so the above refinement implies $mssp(S, t|\tau) \Rightarrow mssp(seq(S, \#t), \tau)$, which is precisely Definition 3.

6 Third iteration

We have not yet specified what the check for task safety entails. We now refine our formal models to prove that a more low-level set of checks, which have feasible hardware implementations, are sufficient to ensure task safety. To do so, we first introduce a number of constraints on the superimposition operator and then refine our SEQ model to incorporate superimposition.

6.1 Superimposition

In this subsection, we persist with an informal view of superimposition, but we now impose a number of constraints on its behavior. In order to do so, we must first refine knowledge about machine state. We introduce an uninterpreted "consistency" operator $\_ \subseteq \_ : \mathcal{S} \times \mathcal{S} \mapsto \{\texttt{true}, \texttt{false}\}$, understanding informally that $S_1 \subseteq S_2$ implies that S_1 is consistent with S_2 in the sense that all of the storage cells of S_1 are also available in S_2 and, further, that both agree on the values held in those cells.

Definition 8 (Superimposition properties). *Superimposition satisfies*

1. *Associativity:* $(S_1 \leftarrow S_2) \leftarrow S_3 = S_1 \leftarrow (S_2 \leftarrow S_3)$*;*
2. *Containment:* $S_1 \subseteq S_2$ *implies* $(S_1 \leftarrow S_3) \subseteq (S_2 \leftarrow S_3)$*;*
3. *Idempotency:* $S_2 \subseteq S_1$ *implies* $(S_1 \leftarrow S_2) = S_1$*.*

6.2 Sequential execution, refined

A sequential machine operates by fetching an instruction, decoding and executing it, and then writing the results back to machine state. This view leads to a definition of instruction execution in terms of superimposition. Before that, however, we must address a problem incurred by SEQ in the context of MSSP slave execution. Tasks evolve by sequentially advancing their live-in sets (as per Definition 5). Since those live-in sets are produced by the master, they are potentially unsuitable for the purposes of executing the next instruction (we make no assumptions about the live-ins produced by the master). To serve as a precondition for well-defined sequential execution, we introduce an uninterpreted notion of machine state *completeness*. Completeness is largely ISA-specific, but

we can understand it in a general sense as follows. A machine state is complete for an instruction's execution if it contains a cell for the program counter, the memory cell pointed to by that program counter (the instruction itself), and all other cells (registers and/or memory locations) that the instruction will read during its execution.

Definition 9 (Instruction execution). *If $S \in \mathcal{S}$ is complete, then we define $next(S)$ to be $S \leftarrow \delta(S)$. Thus, $\delta(S) \in \mathcal{S}$ constitutes the changes to state that will result from executing the next instruction.*

Effectively, the function $\delta : \mathcal{S} \mapsto \mathcal{S}$, which is defined only if its argument is complete, performs the fetch-decode-execute steps alluded to above; the superimposition "commits" the results.

A key property of sequential execution upon which our results depend is *determinism*. Specifically, we require that advancing two *consistent* machine states by the same number of steps must yield consistent results. Formally, $S_1 \subseteq S_2$ must imply $seq(S_1, n) \subseteq seq(S_2, n)$. This can be inferred from the more basic requirement that execution of a single instruction be deterministic: $S_1 \subseteq S_2$ must imply $\delta(S_1) = \delta(S_2)$. That is, two consistent, complete states, which will execute the same instruction on the same data when advanced one step, must produce the same set of outputs.

Of course, sequential execution of n instructions is well-defined only if, at each step along the way, the machine state is complete. When this is the case for a given machine state, we will say that state is n-complete. More formally, S is n-complete if it is complete (for one instruction) and $next(S)$ is $(n-1)$-complete.

In order to define sequential execution in terms of superimposition, we introduce the notion of *cumulative writes*, which are the results that accrue from the sequential execution of multiple instructions.

Definition 10 (Cumulative writes). *The cumulative writes generated by sequential execution are given by $\Delta : \mathcal{S} \times \mathbb{Z}^+ \mapsto \mathcal{S}$. For all $n \geq 0$, we define*

$$\Delta(S, n) = \left\{ \begin{array}{ll} \emptyset & \text{if } n = 0 \\ \Delta(S, n-1) \leftarrow \delta(seq(S, n-1)) & \text{otherwise} \end{array} \right]$$

From properties of superimposition, determinism and cumulative writes we obtain the following important result.

Lemma 3. *For all $n \geq 0$, the following hold.*

- *If $S \in \mathcal{S}$ is n-complete, then $seq(S, n) = S \leftarrow \Delta(S, n)$.*
- *For $S_1, S_2 \in \mathcal{S}$ n-complete, $S_1 \subseteq S_2$ implies $\Delta(S_1, n) = \Delta(S_2, n)$.*

6.3 Establishing task safety

We can now show that checking for task safety, which we have assumed is a basic capability of the MSSP machine, is equivalently performed through two low-level checks. This result is expressed formally as follows.

Theorem 2. *If $S_1 \subseteq S_2 \in \mathcal{S}$ are n-complete then $seq(S_2, n) = S_2 \leftarrow seq(S_1, n)$.*

This result follows from Lemma 3 and the properties of superimposition that we enumerated in Definition 8. Specifically, since S_1 is n-complete, $seq(S_1, n) = S_1 \leftarrow \Delta(S_1, n)$. Hence, $S_2 \leftarrow seq(S_1, n) = S_2 \leftarrow (S_1 \leftarrow \Delta(S_1, n))$. Since superimposition is associative, the right hand side is the same as $(S_2 \leftarrow S_1) \leftarrow \Delta(S_1, n)$. But $S_1 \subseteq S_2$, so $S_2 \leftarrow S_1 = S_2$. Thus, $S_2 \leftarrow seq(S_1, n) = S_2 \leftarrow \Delta(S_1, n)$. But we also know that $\Delta(S_2, n) = \Delta(S_1, n)$, and hence that $S_2 \leftarrow seq(S_1, n) = S_2 \leftarrow \Delta(S_2, n)$. The latter expression is exactly $seq(S_2, n)$.

The obvious implication of this result is that completeness and consistency imply task safety: if S is the architected state of an MSSP machine, and $t \in \mathcal{T}$ is some task such that $live\_in(t) \subseteq S$ and $live\_in(t)$ is $\#t$-complete, then $seq(S, \#t) = S \leftarrow seq(live\_in(t), \#t)$. That is, t is safe for S.

7 Conclusion

We have shown that MSSP achieves the equivalent of a sequential execution, albeit at the coarser granularity of tasks rather than instructions. Through our formalization of its operation, we isolated the concept of *task safety* as the principal factor that underpins correctness. We proved that safety follows from *completeness* and *consistency* of live-ins with respect to architected state, two requirements that the existing MSSP architecture can easily be shown to satisfy.

In establishing the above results, we also discovered a number of unexpected properties of MSSP. Good examples are the associativity of superimposition (commits) and our dependence on determinism in SEQ. In a sense, these results are merely artifacts of the formalization process itself — superimposition's associativity, for example, was needed in the proof of one of our lemmas. In this respect, we feel the process of deriving the formal model was as beneficial to our understanding as was the final model itself. On those grounds alone, the exercise proved its worth.

In general, all the benefits we reaped in this work are a result of the systematic, rigorous thinking necessitated by formalization. The computer architects involved in this work found such rigor particularly liberating because it permitted us to focus on the fundamental, implementation-independent issues, rather than on the intricate performance-mandated design points. Indeed, the ability to separate correctness from performance concerns pervades our work; the formalization of MSSP reinforced our conviction in this respect.

We found the process of mechanizing our proofs in Maude to be easy and intuitive. Deriving the Maude modules from a manual (pencil-and-paper) effort [9] was completed in well under a week, mostly by a novice Maude user. The mechanization did force an even more rigorous approach, which, in turn, exposed even more fundamental assumptions we were making. For example, our discovery that commit order is not important is a good example of how Maude assisted us — having to be explicit about associativity and commutativity of operators brought this issue to the fore. That said, we were on occasion frustrated by the system's inability to reduce certain terms as required, which forced

us to sometimes organize the modules in a non-ideal fashion. In mitigation, this problem eased as our experience grew, but certainly a proof assistant tool, which permits its user to specify explicitly which rewriting rules — be they equational or transitional — should be applied, would have been a boon.

In summary, our efforts in the formal verification of MSSP have been enormously fruitful. In fact, our positive experiences have motivated further work. We have recently started reasoning about MSSP operation on machine state, such as memory-mapped I/O addresses, where we cannot rely on accesses being *idempotent*. Speculative execution is precluded in such regions, demanding that we impose task boundaries and proceed, non-speculatively, as per SEQ.

References

1. Arvind and X. Shen. Using term rewriting systems to design and verify processors. *IEEE Micro*, 9(3):36–46, May/June 1999.
2. J.R. Burch and D.L. Dill. Automatic verification of pipelined microprocessor control. In *Proc. International Conference on Computer Aided Verification*, volume 818 of LNCS, pages 68–80, June 1994.
3. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *Maude 2.0 Manual*, 2003. `http://maude.cs.uiuc.edu/manual`.
4. M. Clavel, F. Durán, S. Eker, and J. Meseguer. Building equational proving tools by reflection in rewriting logic. In *CAFE: An Industrial-Strength Algebraic Formal Method*. Elsevier, 2000.
5. D.L. Dill, A.J. Drexler, A.J. Hu, and C.H. Yang. Protocol verification as a hardware design aid. In *Proc. IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 522–525, October 1992.
6. L. Lamport. What good is temporal logic? In *Information Processing '83: Proc. IFIP 9th World Congress*, pages 657–668, September 1983.
7. J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, April 1992.
8. P. Salverda, G. Roşu, and C. Zilles. Maude formalization of MSSP. `http://fsl.cs.uiuc.edu/mssp`.
9. P. Salverda and C. Zilles. Formal verification of MSSP. Technical Report UIUCDCS-R-2003-2384, University of Illinois at Urbana-Champaign, December 2003.
10. J. Sawada and W.A. Hunt. Trace table based approach for pipelined microprocessor verification. In *Proc. International Conference on Computer Aided Verification*, volume 1254 of LNCS, pages 364–375, June 1997.
11. J. Sawada and W.A. Hunt. Processor verification with precise exceptions and speculative execution. In *Proc. International Conference on Computer Aided Verification*, volume 1427 of LNCS, pages 135–146, June 1998.
12. G. Sohi, S. Breach, and T. Vijaykumar. Multiscalar processors. In *Proc. 22nd Annual International Symposium on Computer Architecture*, pages 414–425, June 1995.
13. C. Zilles. *Master/slave speculative parallelization and approximate code*. PhD thesis, University of Winsconsin - Madison, 2002.
14. C. Zilles and G. Sohi. Master/slave speculative parallelization. In *Proc. 35th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 85–96, November 2002.