Virtual Multiprocessor: An Analyzable, High-Performance Microarchitecture for Real-Time Computing*

Ali El-Haj-Mahmoud, Ahmed S. AL-Zawawi, Aravindh Anantaraman, and Eric Rotenberg
Department of Electrical and Computer Engineering – Center for Embedded Systems Research
North Carolina State University, Raleigh, NC 27695
{aaelhaj,aalzawa,avananta,ericro}@ncsu.edu

ABSTRACT

The design of a real-time architecture is governed by a trade-off between analyzability necessary for real-time formalism and performance demanded by high-end embedded systems. We reconcile this trade-off with a novel *Real-time Virtual Multiprocessor* (RVMP). RVMP virtualizes a single in-order superscalar processor into multiple interference-free different-sized *virtual processors*. This provides a flexible spatial dimension. In the time dimension, the number and size of virtual processors can be rapidly reconfigured. A simple real-time scheduling approach concentrates scheduling within a small time interval, producing a simple repeating space/time schedule that orchestrates virtualization. RVMP successfully combines the analyzability (hence real-time formalism) of multiple processors with the flexibility (hence high performance) of simultaneous multithreading (SMT).

Worst-case schedulability experiments show that more task-sets are provably schedulable on RVMP than on conventional rigid multiprocessors with equal aggregate resources, and the advantage only intensifies with more demanding task-sets. Run-time experiments show RVMP's statically-controlled coarser-grain space/time configurability is as effective as unsafe SMT. Moreover, RVMP provides a real-time formalism that SMT does not currently provide.

Categories and Subject Descriptors

C.3 [**Special-Purpose and Application-Based Systems**]: *Real-time and embedded systems*; C.1.3 [**Processor Architectures**]: Other Architecture Styles—*pipeline processors*

General Terms

Design, Performance

Keywords

Simultaneous multithreading, superscalar processor, resource partitioning, hard real-time, worst-case execution time, scheduling

*This research was supported in part by NSF CAREER grant CCR-0092832, NSF grants CCR-0207785, CCR-0208581 and CCR-0310860, and generous funding and equipment donations from Intel and Ubicom.

CASES'05, September 24–27, 2005, San Francisco, California, USA.

1. INTRODUCTION

To meet higher performance targets, high-end general-purpose embedded processors have inherited microarchitecture features from their desktop counterparts, including deep pipelining, dynamic branch prediction, multiple instruction issue, multithreading, and even out-of-order execution. For example, ARM11-derived processors have an 8-stage pipeline with dynamic branch prediction and caches [8], Ubicom's IP3023 supports 8 hardware threads [40], and IBM's embedded PowerPC 750 [20] is a dynamically-scheduled 2-way superscalar processor.

Unfortunately, dynamic performance-enhancing techniques (most notably dynamic branch prediction, caches, out-of-order execution, and dynamic multithreading) make it difficult to design a large class of embedded systems, hard-real-time systems. A real-time system typically runs multiple periodic tasks (collectively called a task-set), where each task repeats at fixed time intervals equal to the period of the task (Figure 1 shows an example task-set). For tractable analysis, a task's deadline is simply its period [31], that is, an instance of a task (*e.g.*, A1 in Figure 1) must always complete before the next instance of the task (*e.g.*, A2 in Figure 1) is released. Guaranteeing this criterion for all tasks in the task-set guarantees the task-set is schedulable as a whole (the system will never be overrun). Schedulability of a task-set must be proven or disproven *a priori*, using only the worst-case execution times (WCETs) and periods of tasks. Despite some advances in worst-case timing analysis, in practice, deriving tight and safe (provably never exceeded) WCETs of tasks on processors with dynamic branch prediction, caches, and out-of-order execution, *etc.*, is intractable. As such, dynamic microarchitecture features significantly complicate and even undermine the design process of these systems [17, 1].

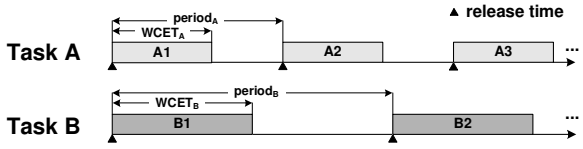


Figure 1: Periodic task model.

Higher performance can be achieved without sacrificing analyzability, either by increasing the frequency of a simple processor via deeper pipelining or by using multiple simple processors. Multiple simple processors is an attractive solution due to contemporary integration capability and multiprocessor-system-on-chip (MpSOC) trends, combined with the natural availability of multiple tasks in typical real-time embedded systems. However, the rigid and uniform partitioning of resources among multiple processors leads to load-balance problems, which may cause demanding task-sets to be artificially unschedulable. That is, sufficient resources may be available in aggregate, but individual tasks cannot be spread

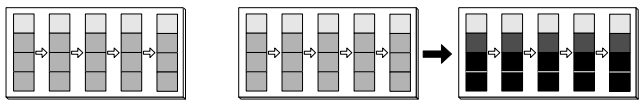
across multiple processors. Alternatively, the system could be over-designed with more processors to compensate.

A more flexible substrate for resource sharing is preferred, to better utilize aggregate resources and improve the cost-performance metric. Simultaneous Multithreading (SMT) [39] meets a similar need in server and desktop systems, enabling fine-grain resource sharing among multiple threads for higher overall system throughput. However, as with other dynamic techniques described earlier, SMT undermines the analytical foundation of hard-real-time scheduling. Because there is interference among simultaneous tasks, the WCET of a task must now be derived in the context of other tasks in the task-set. This is contrary to conventional worst-case timing analysis, which assumes a task runs alone on the processor, and as such derives WCETs of tasks separately. Moreover, deriving WCETs of multiple tasks running together on an SMT processor is intractable. Since tasks have different periods, specific task combinations vary over time, as does the amount by which co-scheduled tasks overlap. Even if we know which tasks are running and which of their regions overlap, instructions from different tasks dynamically compete for shared processor resources. To sum up, SMT is incompatible with proving hard-real-time guarantees.

In this paper, we reconcile the trade-off between performance and analyzability in real-time systems, by introducing a *Real-time Virtual Multiprocessor* (RVMP) system. RVMP is the combination of an analyzable, high-performance microarchitecture and a simple real-time scheduling approach.

1.1 RVMP architecture

Our novel architecture combines the analyzability of multiple dedicated processors with the flexible resource sharing (hence higher performance and favorable cost-performance) of SMT. We propose a highly reconfigurable multithreaded superscalar processor that provides two levels of flexibility, in *space* and *time*. In the space dimension, the processor's resources can be arbitrarily partitioned to create multiple dedicated virtual processors, with possibly different performance levels according to the resource partitioning. Multiple tasks execute at the same time, one on each partition, without interfering with each other. Interference-free partitions achieve the necessary isolation for analyzability, like a conventional multiprocessor. Yet, because different-sized partitions can be carved out of aggregate resources of the single superscalar processor underneath, we overcome schedulability limitations of multiple equal-sized processors. Superscalar "ways" (for example, there are 4 ways in a 4-way superscalar processor) present a natural resource partitioning strategy. For example, Figure 2(a) shows two interference-free partitions, one composed of 1 way and the other of 3 ways. In the time dimension, the resource partitions can be rapidly reconfigured, even every cycle, fluidly changing the number and size of partitions if so desired. For example, Figure 2(b) shows the same two interference-free partitions being reconfigured into three interference-free partitions, two composed of 1 way each and one composed of 2 ways. When and how the partitions are adjusted is determined by a static schedule, generated by our novel real-time scheduling framework.



(a) **(b)**

Figure 2: Processor space/time partitioning example. (a) Space partitioning. (b) Rapid reconfiguration.

A crucial contribution of RVMP's *interference-free* approach is that it preserves single-task WCET analysis. That is, the WCET of a task can still be derived independent of which other tasks are co-scheduled with it, thanks to interference-free partitions.

Regarding the underlying superscalar processor from which partitions are carved, its complexity is only limited by what WCET analysis tools can handle. Currently, dynamic techniques are beyond the capabilities of most WCET analysis methods. Accordingly, in this paper, the underlying processor issues instructions in order and uses static branch prediction and software-managed scratchpad memories (instead of caches). In-order issue does not significantly impact the performance of RVMP. Decoupled virtual processors allow for arbitrary slippage among independent threads, creating an implicit out-of-order execution among different threads. As such, the performance gain from thread-level parallelism offsets the performance loss due to in-order issue within threads, corroborated by others in the context of in-order SMT processors [19]. Programmatic memory transfers between main memory and on-chip scratchpad memory are often used in hard-real-time applications for determinism, not to mention possibly better performance due to programmer/compiler managed layout [41]. It has been shown that static branch prediction actually interacts favorably with WCET analysis, whereas dynamic branch prediction often works against it. Statically predicting the longest path yields a safe WCET and also the tightest possible WCET, by virtue of adding the misprediction penalty to what is the shorter path anyway [3].

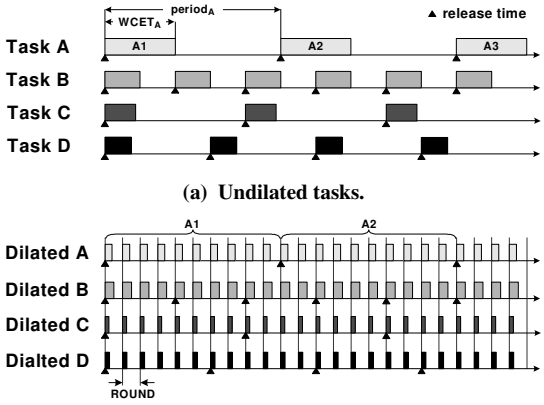
We use the high-performance Alpha 21164 4-way in-order superscalar processor [12] as a starting point, augmented with replicated register files and program counters to support multiple simultaneous threads like the Ubicom IP3023 embedded processor [40]. We propose novel pipeline extensions for aggregating individual ways in both the processor's front-end (fetch, decode, and issue ways) and back-end (multiple heterogeneous execution pipelines), to form one or more interference-free partitions. Forming interference-free partitions is not always as literal as what is physically implied by the high-level examples shown in Figure 2. The novelty of our pipeline extensions lies in achieving the effect of physically distinct different-sized partitions, even when the separation is not physically apparent in certain pipeline stages. Novel mechanisms include:

- A new fetch buffer design facilitates assembling a pre-determined number of instructions for each virtual processor every cycle. This design minimizes impact on the critical instruction fetch unit itself, namely we avoid multiple configurable-width I-scratchpad ports.
- The Alpha 21164 in-order issue stage includes the "slotter" and "scoreboard" logic, responsible for checking data and structural hazards and steering ready instructions to respective execution pipelines. First, we show that the steering datapath is unchanged since all issue slots are connected to all execution pipelines in any case. Second, we identify natural "intervention" points in the control logic for easily decoupling issuing among different virtual processors.
- While our Alpha derivative pipeline fully replicates some function units – *e.g.*, there is a simple integer unit in each of the four execution pipelines – certain function units are only available in some of the execution pipelines (*e.g.*, floating-point units, agen units/D-scratchpad ports). These may need to be shared among multiple virtual processors, seemingly violating the interference-free requirement. This is addressed by conservatively time-multiplexing shared function units, again dictated by the static schedule mentioned earlier. This increases the perceived latency every time the shared resource is used, but determinism and overall performance benefits outweigh this localized slowdown.

- Reconfiguring the number and size of partitions often coincides with changing which hardware threads are currently using partitions, again determined by the static schedule. This is not a context switch, just a change in thread selection. However, unlike SMT thread selection, the interference-free requirement stipulates that a deselected thread must appear to instantly relinquish its entire partition before reconfiguring the processor. The processor back-end is not a problem since the execution pipelines are non-blocking – already-issued instructions are allowed to finish. Two small shadow buffers (64 bytes each) connected to the blocking decode and issue stages facilitate physically moving the deselected thread's unissued instructions. Moreover, the shadow buffers facilitate moving the preempted instructions back again when the previous configuration is restored. (The new fetch buffer design is inherently non-blocking and does not require a separate shadow buffer.)

1.2 RVMP scheduling

We develop a new real-time scheduling framework that specifically capitalizes on the architecture's space/time features, (1) arbitrary interference-free partitions and (2) rapid reconfiguration, to yield a scheduling approach that is simple yet highly effective. In turn, the architecture is orchestrated by a simple static schedule produced by the scheduling framework.



(a) Undilated tasks.

(b) Dilated tasks.

Figure 3: Task-set.

A task-set consists of multiple tasks with different worst-case execution times (WCETs) and periods. For example, Figure 3(a) shows a task-set with four tasks. New instances of a task are released based on the task's period, as explicitly highlighted for task A (instances A1, A2, and A3) in Figure 3(a). Generating a static schedule (if a feasible one exists) involves considering arbitrary space-sharing and time-sharing of the new architecture among tasks. However, scheduling is impractical for the task-set as shown. Because tasks have different periods, the schedule repeats only after an entire "hyper-period", the least-common-multiple of all tasks' periods (too long to show in Figure 3(a)). The hyper-period may be millions of cycles or more, depending on the task-set. Within such a long time span, there is an overwhelming number of possible space/time configuration sequences that must be searched to find a feasible schedule. Moreover, the dedicated-hardware cost is high, in terms of storing a lengthy static space/time schedule.

Our solution to this problem is to "spread" out each instance of a task throughout its period, for all tasks, as shown in Figure 3(b). We define a small interval of time, say 100 cycles, called a *round*. Each task runs for only a fraction of the round, then it is temporarily suspended for the remainder of the round, and then it is resumed in the next round. This process repeats indefinitely, since the completion of a task's dilated instance (*e.g.*, A1 in Figure 3(b)) meets with the release of its next instance (*e.g.*, A2 in Figure 3(b)), intentionally. The nice effect is to create a "sub-period" common to all tasks. That is, the schedule repeats every round. Therefore, we only need to concentrate on scheduling *a single round* instead of the entire hyper-period.

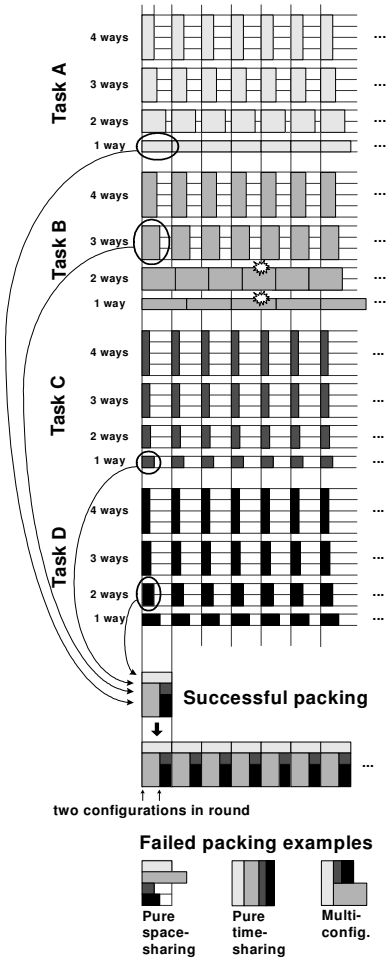


Figure 4: Packing of dilated tasks.

Our round-based scheduling framework considers fractional WCETs of each task on 4, 3, 2, or 1 superscalar ways, as shown in Figure 4. We use a standard bin-packing algorithm [7] to try to fit fractional tasks into one or more architecture configurations within a round, ranging from pure time-sharing (every task uses all 4 ways, sequentially) to pure space-sharing (all tasks run at the same time, one per way). Cases in between pure time-sharing and pure space-sharing involve multiple architecture reconfigurations in the round. Three example failed bin-packing attempts are shown at the bottom of Figure 4, pure space-sharing, pure time-sharing, and a multi-configuration schedule. The successful bin-packing attempt produces a two-configuration round: (1) task A on 1 way and task B on 3 ways, for 60 cycles, followed by (2) task A on 1 way, task C on 1 way, and task D on 2 ways, for 40 cycles. Thus, the processor is reconfigured twice per round. A task-set is considered unschedulable on the architecture if bin-packing fails to find a feasible schedule

(in which case the system designer needs to revise task periods, optimize tasks' WCETs, and/or consider higher frequency or different processors, *etc.*).

To sum up, our scheduling approach is simple because we only need to statically schedule a small time interval, the round. The static schedule repeats indefinitely as shown in Figure 4. The efficient static schedule is stored in a compact hardware table that controls the processor partitioning.

1.3 Contributions and paper outline

This paper makes the following major contributions.

1. *RVMP architecture*. We propose hardware mechanisms for virtualizing a single superscalar processor into multiple different-sized virtual processors. The virtual processors are truly interference-free despite their creation from a common processor underneath. The architecture presents the analyzability of multiple dedicated processors with the flexibility of SMT. Interference-free virtual processors provide the isolation needed for tractable analysis (both in terms of preserving single-task WCET analysis and facilitating real-time scheduling), thus inheriting the analyzability of multiple dedicated processors. On the other hand, different-sized virtual processors and rapid reconfiguration emulate flexible resource sharing of SMT.
2. *Real-time scheduling framework for the RVMP architecture*. We propose a real-time scheduling framework that interacts closely with the architecture, yielding a scheduling approach that is both simple and effective. Dilating tasks throughout their periods enables scheduling to be concentrated within a small interval of time, the round. In stark contrast to the alternative of scheduling an entire hyper-period, round-based scheduling is tractable, the round's time span is task-set-independent, and storing the compact static schedule for a round is inexpensive.
3. *Key performance comparisons with rigid MPs and unsafe SMTs*. We demonstrate that schedulability of task-sets is not substantially affected by excluding out-of-order execution within tasks for analyzability. Implicit out-of-order execution among tasks on different virtual processors – a source of decoupling that does not undermine schedulability analysis – compensates for in-order execution within tasks. This is observed experimentally: RVMP provably schedules task-sets (*i.e.*, for all possible inputs) that pass dynamic testing with *specific inputs* on an unsafe but otherwise equivalent conventional SMT processor. Moreover, RVMP successfully schedules task-sets that are not schedulable on a rigidly-partitioned multiprocessor with equal aggregate resources.
4. *Hierarchical classical/RVMP scheduling*. RVMP naturally supports task-sets which have more tasks than the architecture has virtual processors, by assigning multiple tasks to each virtual processor and applying classical uniprocessor scheduling policies (*e.g.*, earliest-deadline-first or fixed-priority [30]) to schedule tasks that share a virtual processor. The key generalization for multiple tasks per virtual processor: dilation is based on the combined utilization of tasks sharing a virtual processor [13]. As with conventional multiprocessors, determining which tasks to assign to the same virtual processor adds another dimension to scheduling. We first show that conventional multiprocessor assignment approaches extend to RVMP. We then highlight an RVMP-specific dimension, namely that it is beneficial to specifically consider assigning tasks with similar way-preferences to the same virtual processor, since the virtual processor will ultimately receive a single choice of issue width.

The rest of this paper is organized as follows. The RVMP architecture and RVMP real-time scheduling framework are described in Sections 2 and 3, respectively. Section 4 outlines our experimental methodology. Results are presented in Section 5. Related work is discussed in Section 6. We summarize the paper in Section 7.

2. REAL-TIME VIRTUAL MULTI-PROCESSOR (RVMP) ARCHITECTURE

The RVMP architecture is built on top of an in-order superscalar processor. The Alpha 21164 [12], an in-order 4-way superscalar processor, serves as a good starting point, partly because of its high-performance emphasis and partly because of available documentation (including an interesting description of its hierarchical issue logic). The RVMP processor architecture is shown in Figure 5.

Unlike the single-threaded 21164, RVMP supports 4 thread contexts, namely 4 program counters and 4 copies of the integer and floating-point register files. Each hardware thread corresponds to one *virtual processor* (VP). Thus, in this paper, there are 4 VPs.

Software-managed instruction scratchpad (I-scratchpad) and data scratchpad (D-scratchpad) memories are used instead of caches for deterministic high-performance. The I-scratchpad is interleaved, having two single-ported banks to guarantee fetching four sequential instructions from a single thread every cycle. The D-scratchpad has one read port and one read/write port, supporting issuing up to two loads or one load and one store per cycle.

The processor has four integer execution pipelines, $FU0$ (simple integer), $FU1$ (simple integer and integer multiplication/division), $FU2$ (simple integer and load/store address generation), and $FU3$ (simple integer and load/store address generation). There is one floating-point execution pipeline, $FU4$. All function units are pipelined and can accept new instructions every cycle.

Key modifications are made to the fetch and issue stages to achieve the effect of multiple different-sized interference-free VPs, emulating the simplified depiction of Figure 2. Light-gray shading in Figure 5 highlights the modified fetch and issue stages, discussed in Sections 2.1 and 2.2, respectively.

Dark-gray shading in Figure 5 highlights new structures. Two sets of shadow buffers are coupled to the decode and issue stages to support rapid reconfiguration of the processor (Section 2.3). Virtualization is orchestrated by a hard-real-time table (HRT), which contains a static schedule of the processor resources for a single round (Section 2.4).

2.1 Instruction fetch

Each cycle, the instruction fetch stage must supply a certain number of instructions from each configured thread, based on the widths of their corresponding partitions. Moreover, these instructions must be aligned with their corresponding partitions in the subsequent decode and issue stages. For example, for the configuration in Figure 2(a), the fetch stage must assemble four instructions every cycle for the decode stage, comprised of one instruction from the 1-way thread followed by three instructions from the 3-way thread. Making the I-scratchpad arbitrarily partitionable requires multiple interference-free configurable-width ports, an expensive prospect in terms of area and complexity. Instead, we transfer the instruction assembly functionality to a custom fetch buffer, concentrating complexity within a more scalable structure.

The custom fetch buffer serves as a translation mechanism between (1) the single wide fetch port of the I-scratchpad that clearly favors time-sharing and (2) multiple narrower partitions in the space-shared decode/issue stages. A thread's instructions are fetched from the I-scratchpad in individual bursts of (at most) 4 instructions, into a dedicated 4-instruction column of the fetch buffer. Then, the fetch buffer drains and aligns instructions from the column at an even pace that matches the width of the corresponding partition.

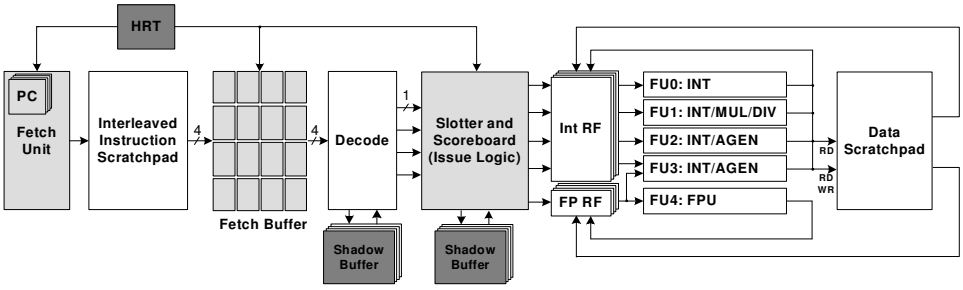


Figure 5: RVMP processor architecture.

In more detail:

- In a given cycle, the I-scratchpad delivers (at most) 4 instructions from one thread corresponding to a configured VP. Each VP has a dedicated 4-instruction column in the fetch buffer. Thus, since there are four VPs, there are four columns. The (at most) 4 fetched instructions are written into the corresponding column. Since, in a given cycle, instructions are fetched on behalf of only one VP, the fetch buffer only requires a single 4-instruction-wide write port to write a whole column.
- To maintain equilibrium between incoming/outgoing instructions from each VP's column, VPs are assigned fetch cycles commensurate with their partition widths. That is, 1-way, 2-way, 3-way, and 4-way VPs are assigned 25%, 50%, 75%, and 100% of fetch cycles, respectively. For example, a 2-way VP inserts (at most) 4 instructions into its column every other cycle. Assuming no stalls in later stages of the VP, 2 instructions are removed from its column in each of two consecutive cycles, in time for the next 4-instruction insertion. If a minimum of one cycle is needed between insertion and removal, at most 4 additional instruction slots per column accommodates all scenarios in which draining is not fully caught up by the time of the next insertion. (This definitely is required for a 3-way VP, which inserts 4 instructions in three out of four cycles: consecutive 4-instruction insertions temporarily outrun consecutive 3-instruction removals.)
- The fetch buffer has four 1-instruction-wide read ports. Each read port can access any instruction in the fetch buffer. Independent fine-grain read ports provides arbitrary configurability, in terms of assembling a certain number of instructions for each configured thread and aligning them with the corresponding partitions in the subsequent decode and issue stages.

The fetch buffer plays another vital role. When a VP is suspended during the round (due to reconfiguration), its fetched instructions remain in the corresponding fetch buffer column until the VP is resumed during the next round. Per-VP storage in the fetch stage, plus shadow buffers coupled to the decode and issue stages (discussed in Section 2.3), gives each VP the ability to (1) instantly suspend without blocking progress of other VPs and (2) instantly resume from the point it was suspended, preserving the integrity of assumed-suspension-free WCET bounds.

2.2 Instruction issue

We first briefly explain how instruction issuing works in the single-threaded 21164, as best we can infer from a detailed paper [12]. Then, we discuss three key intervention points in the issue logic that are exploited to achieve the effect of interference-free partitions.

2.2.1 Background on 21164 issue logic

The 21164 issues instructions strictly in program order. Instruction issue is implemented in two phases, the slot logic and scoreboard logic. The two phases implement hazard resolution hierarchically, first resolving hazards within a fetch/decode group (slot logic) and then resolving hazards between the fetch/decode group and already-issued instructions (scoreboard logic).

The slotter consists of a 4-instruction staging buffer and a crossbar for steering instructions from the staging buffer to the execution pipelines. Four instructions received from the decode stage are placed in the staging buffer in program order. Since the 21164 is single-threaded, all four instructions belong to the same thread. The purpose of the first phase is to detect data dependences and conflicts for execution pipelines, only among instructions in the staging buffer. Only a contiguous block of independent and non-conflicting instructions, starting from the oldest instruction in the staging buffer and stopping at the first dependent or resource-conflicting instruction, are declared as "ready" for advancement to the second phase. These preliminarily ready instructions are steered to the appropriate execution pipelines, where the second phase begins.

Instructions that advance from the staging buffer to the execution pipelines check the register scoreboard before issuing. The scoreboard detects read-after-write and write-after-write hazards between instructions in the issue stage and instructions already in the execution pipelines. When a hazard is detected, the instruction is prevented from issuing to the register read stage and the function unit. Independent instructions that had advanced with it from the staging buffer, which are logically after the instruction in program order, are also stalled from issuing.

2.2.2 RVMP issue logic

In RVMP, the datapath associated with the staging buffer does not need to be changed. As before, the crossbar facilitates steering any instruction in the staging buffer to any execution pipeline. For decoupling issuing among different VPs, we identify three key intervention points in the control logic to work as seamlessly as possible with the existing control logic.

The staging buffer may contain instructions from multiple VPs. Fortunately, each VP will have its instructions in contiguous staging buffer entries, in program order, as assembled by the fetch buffer. Above, we highlighted the logic that checks for dependences among instructions in the staging buffer. This logic compares the source operands of newer instructions in the staging buffer with the destination operands of all older instructions. Within the staging buffer, the physical arrangement of instructions matches their program order and presumably the dependence checking logic is hardwired accordingly. The match between physical and logical order is preserved within partitions. Therefore, the hardwired dependence checking logic is compatible with multiple partitions. We only need to include VP IDs in the operand comparisons, thereby partitioning the dependence checking logic among VPs, decoupling the first phase of instruction issuing.

The staging buffer control logic also checks for execution pipeline conflicts among instructions in the staging buffer. In RVMP, conflicts among instructions from different VPs are prevented statically via the HRT (Section 2.4). The HRT specifies the owner (VP) of each execution pipeline every cycle. Thus, the second key intervention point is overriding per-instruction request signals in the staging buffer with ownership information from the HRT. By the same token, conflicts are resolved the same as before for instructions in the same VP: one or more instructions in the same VP may request an execution pipeline if their VP owns it this cycle, initiating arbitration as before.

Finally, intervention is also needed in the second phase, the scoreboard. Since the multithreaded processor has per-thread register files, it naturally requires per-thread scoreboards. Instructions use their VP IDs to lookup the corresponding scoreboard. A stalled instruction only causes other instructions in the same VP to stall. This is achieved by gating stall signals with VP IDs.

2.3 Shadow buffers

Reconfiguring the processor involves suspending one or more of the currently configured VPs and resuming one or more suspended VPs, as shown in Figure 2(b). Tasks' WCETs are derived conventionally, *i.e.*, without knowledge of round-based suspend/resume operations. This means round-based suspend/resume operations must have no perceived execution time overhead (or at least a known worst-case overhead, preferrably small, that can be added to tasks' WCETs).

The problem is, at the time of reconfiguration, a newly suspended VP still has instructions in the pipeline. If these instructions are stalled, they will block newly resumed VPs, violating interference-free requirements (single-task WCETs are no longer provably safe). Moreover, instructions of the newly suspended VP will be confused for instructions of one or more newly resumed VPs (as old partitions are repartitioned).

We only need to consider pipeline stages that may block, namely stages in the processor frontend (fetch, decode, and issue). The execution pipelines are free-flowing, so it is safe to allow already-issued instructions of newly suspended VPs to finish. Although the fetch stage is blocking, VPs cannot block each other thanks to dedicated storage per VP in the custom fetch buffer (columns).

We couple a set of shadow buffers to each of the decode and issue stages, to checkpoint/restore the contents of the stages across processor reconfigurations. The number of shadow buffers per set is the same as the number of VPs, since there is a maximum of #VP reconfigurations per round (pure time-sharing). When the configuration of the processor is changed, the four instructions in the decode stage and the four instructions in the issue stage are saved to one of the shadow buffers in each set. During the next round, at the beginning of the same configuration, those instructions are placed again in the corresponding stage latches. Each set of shadow buffers requires only 64 bytes of storage (4 buffers $\times$ 4 instr. $\times$ 4 bytes).

The shadow buffers are extensions of the pipeline and as such do not pose any unusual problems regarding intrerrupt handling. If a VP is interrupted, instructions of the corresponding thread must be drained from the pipeline whether or not the VP is currently suspended. The design of any real-time system, conventional or RVMP-based, requires bounding the worst-case interrupt handling latency as well as bounding the worst-case number of interrupts.

2.4 Hard real-time table (HRT)

The HRT orchestrates the resource sharing among VPs. In particular, the HRT (1) allocates fetch bandwidth, by indicating which VP owns the fetch unit and I-scratchpad each cycle, (2) partitions superscalar ways among VPs, by controlling the assembly of instructions from the fetch buffer (number of instructions from each VP and their alignment with corresponding partitions in the decode/issue stages), and (3) allocates execution pipelines, by specifying the owner VP of each function unit every cycle.

The HRT contains the processor's resource schedule for a single round, as determined by static real-time analysis (Section 3). Each entry of the HRT represents a different processor configuration during the round. Since there is a maximum of #VP reconfigurations per round (pure time-sharing case), the maximum number of HRT entries required is also #VP (four in this paper). For each configuration (*i.e.*, HRT entry), sharing patterns are encoded using a 4-cycle "mini" schedule for each shared resource. The mini schedules specify the resource bandwidth allocated to each VP, as we explain shortly. Each entry consists of:

- An 8-bit lifetime counter (LTC). The LTC indicates the number of cycles per round for which this entry (configuration) is valid. The sum of the LTCs of all entries equals the duration of the round.
- A 4-entry fetch vector (FV). The FV is a 4-cycle cyclic schedule for the fetch unit, specifying which VP to fetch instructions for in each cycle, for the lifetime of the HRT entry. A 4-cycle schedule is enough to specify the percentage of fetch cycles assigned to each VP based on its number of ways.
- A 4-entry partitioning vector (PV). The PV is used, for the lifetime of the HRT entry, to determine how superscalar ways are partitioned among VPs. The PV controls assembly of instructions from the fetch buffer corresponding to partitions in the decode and issue stages.
- Five configuration vectors (CVs), one for each function unit. Like the FV, each CV is a 4-cycle cyclic schedule for the function unit. Each entry of a CV indicates which VP owns the function unit during the corresponding cycle.
- A 2-bit cycle count (CC). During the lifetime of an HRT entry, its CC is used to index (i) the FV, to determine which VP owns the fetch unit in the current cycle, and (ii) all five CVs, to lookup which VP owns which function units in the current cycle. The CC is incremented every cycle, wrapping back to zero every fourth cycle. Thus, the 4-cycle sharing patterns specified by the FV and CVs are repeated every four cycles for the lifetime of the entry.
- A 1-bit end-of-table (EOT) flag. The EOT flag is set for the last valid entry of the table.

The HRT is initialized by software before starting a task-set (*e.g.*, system startup). The total required storage space of the HRT is less than 40 bytes.

Initially, a watchdog counter is loaded with the content of the LTC of the first HRT entry (the active entry). The FV, PV, and CVs of that entry are used to configure the processor. The watchdog counter decrements by one each cycle. When the watchdog counter reaches zero, the next HRT entry becomes the active entry, and so on. When the end of the table (EOT flag) is reached, the active entry wraps back to the first HRT entry, corresponding to the beginning of a new round.

Example. Figure 6 shows the HRT contents for the example partitioning of Figure 4. The static schedule for one round is repeated here for convenience (left-hand side of figure). Recall from that example, there are four tasks in the task-set and each task is mapped to one of the four available VPs. The duration of the round is 100 cycles. There are two different configurations during the round, one active for 60 cycles and the other for 40 cycles. Thus, the HRT contains only two valid entries (the last two entries of the HRT are invalid).

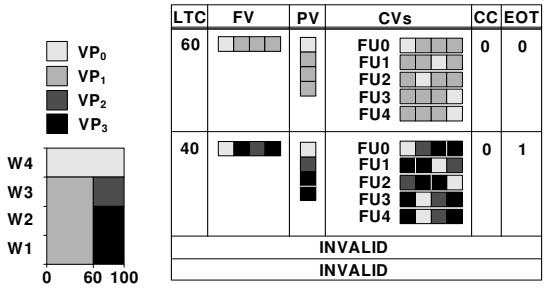


Figure 6: Example HRT contents.

The first entry of the HRT indicates that, for 60 cycles, there are two active VPs: VP_0 and VP_1. The superscalar ways are partitioned between the two VPs as indicated by the PV: 1 way for VP_0 and 3 ways for VP_1. The FV determines the nature in which the two VPs time-share the fetch unit: 1 fetch cycle for VP_0 followed by 3 fetch cycles for VP_1, and so on (thus, VP_0 fetches a peak of 4 instructions every 4 cycles, or 1/cycle, and VP_1 fetches a peak of 12 instructions every 4 cycles, or 3/cycle).

The CVs indicate that the 1-way VP_0 owns each function unit for only 1 cycle out of 4 (25% share) and the 3-way VP_1 owns each function unit for 3 cycles out of 4 (75% share). Consider instructions that can execute in any of the four function units $FU0$-$FU3$, namely simple integer instructions. While VP_0 owns each of $FU0$-$FU3$ only 25% of the time, since there are four of them, one of them will be available each cycle for VP_0. Likewise, three of them will be available each cycle for VP_1. Thus, as should be the case, there are no conflicts for simple integer units and no corresponding impact on tasks' WCETs. On the other hand, consider instructions that have a limited number of function units to choose from. For example, integer multiply instructions can only execute on $FU1$. VP_0 owns $FU1$ for one cycle out of four. Thus, WCET analysis safely extends the latency of multiply instructions in VP_0 by three cycles, the worst-case wait time for a ready multiply instruction in VP_0. Similarly, VP_1 owns $FU1$ for three cycles out of four, and WCET analysis safely extends the latency of multiply instructions in VP_1 by one cycle, the worst-case wait time for a ready multiply instruction in VP_1. To sum up, static arbitration for contended units, via the HRT's CVs, makes it possible to bound the worst-case wait time of instructions that use these units.

The second HRT entry in Figure 6 indicates that three VPs are active for the remaining 40 cycles of the round: VP_0 (1 way, 25% share), VP_2 (1 way, 25% share), and VP_3 (2 ways, 50% share).

3. REAL-TIME SCHEDULING ANALYSIS

Static real-time scheduling analysis for RVMP is responsible for assigning tasks to VPs and allocating processor resources to VPs (in both space and time) to guarantee that tasks will meet their deadlines. The final output is the HRT contents, corresponding to the space/time schedule of VPs within a round.

We first consider the case in which the number of tasks in the task-set is less than or equal to the number of VPs, thus, only a single task is assigned to each VP (Section 3.1). We then extend the framework to cover the general case of multiple tasks per VP (Section 3.2).

3.1 Single task per virtual processor

The analysis attempts to find the least possible space share (number of ways) and time share for each VP within the round, such that tasks assigned to the VPs will meet their deadlines. Strictly speaking, the analysis does not need to find the most efficient schedule, just the first one that works. Nonetheless, by finding the most efficient schedule, excess resources may be used later to attempt scheduling another periodic hard-real-time task, sporadic (one-time) hard-real-time tasks, periodic soft-real-time tasks, *etc*.

The analysis proceeds in two steps. First, the analysis produces a space/time schedule for VPs within the round (unless no feasible schedule is found, in which case the task-set is considered unschedulable on the architecture). Second, the contents of the HRT corresponding to the schedule are synthesized.

3.1.1 Generating space/time schedule

Recall that our analysis is based on evenly spreading out the execution of every task over multiple rounds between their releases and deadlines (Figure 4). This conveniently enables us to concentrate scheduling within a single round.

Each task is guaranteed a fixed fraction of the round, called a *duty cycle* (d, where $d \leq 1$). Since the maximum fraction of time that a task i uses the system *overall* is $U_i = \frac{WCET_i}{period_i}$ (called worst-case utilization), this is naturally the same fraction of the round that task i must be guaranteed. That is, a task's duty cycle is simply its worst-case utilization: $d_i = U_i = \frac{WCET_i}{period_i}$. Since a task's WCET depends on the number of superscalar ways allocated to the VP to which the task is assigned, the task's duty cycle also depends on the way allocation of its assigned VP.

Since the analysis considers a single round and abstracts the processor's resources as superscalar ways, the scheduling algorithm works on a two-dimensional region with area of $R \times W$, where R is the duration of the round (time dimension) and W is the total number of superscalar ways (space dimension). The space/time allotments of VPs are also modeled as two-dimensional regions, each with an area of $(d \times R) \times w$, where w is the number of ways allocated to the VP, d is the duty cycle of the task assigned to the VP assuming w ways, and R is the duration of the round.

The scheduling algorithm considers all possible way allocations (1, 2, 3, or 4 ways) for every VP. For each combination of VPs, we sum the VPs' "areas" (the area of a VP is $(d \times R) \times w$ as explained above). If this sum is greater than the total area available ($R \times W$), the combination is discarded right away because it is impossible to schedule.

Now we need to concentrate on combinations that yield a combined area less than the total available area. For each such combination, we need to fit all the VPs (with their specified superscalar ways and duty cycles) within the overall $R \times W$ region. This is a 2-dimensional bin-packing problem [11]. The bin is a rectangle of width R (duration of round) and height W (total number of superscalar ways). The items we need to pack are the VPs, each of width $d \times R$ (the duration of its duty cycle assuming w ways) and height w (the number of ways allocated to it). Bin packing is an $\mathcal{NP}$-hard problem [11], and there exists a wide range of approximate solutions. Each solution consists of a pre-heuristic, which deals with the order in which the items are packed, and a heuristic, which deals with the packing algorithm itself. The most widely used pre-heuristics are sorting the items (largest first) according to their height, width, area, or perimeter [11]. We use sorting based on perimeter as our pre-heuristic. As for the heuristic, there are plenty of algorithms described in the literature. We use the *Bottom-Left-Fill* (BLF) algorithm [7] in this paper.

The BLF algorithm takes the first item in a sorted list, and finds the bottom- and left-most corner of the bin where the item can fit, and places the item there. This process is repeated until all the items are packed. An example is shown in Figure 7. Assume we have four VPs to which tasks A, B, C, and D are assigned, with way alloca-

tions as shown. The tasks are characterized as follows (Task(duty cycle,ways)): A(1,1), B(0.6,3), C(0.4,1), and D(0.4,2). Since the pre-heuristic sorts based on perimeter, the sorted list is as follows: B, A, D, C. The BLF algorithm starts with an empty rectangular area of $R \times W$ as in Figure 7(a). The first item in the sorted list is task B. The algorithm locates the bottom- and left-most corner, indicated by ($\times$) in Figure 7(a), and places B there. The next item is task A, and two corners are located in Figure 7(b). Task A will fit only in the upper one, so it is placed there (Figure 7(c)). The same procedure is repeated for tasks D (Figure 7(d)) and C (Figure 7(e)).

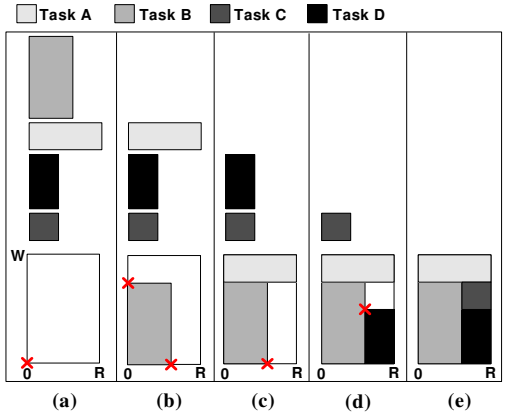


Figure 7: BLF packing example.

Bin packing is repeated for all partitioning combinations (*i.e.*, trying 1, 2, 3, and 4 ways for every VP) that meet the maximum area requirement. Among combinations that succeed the packing algorithm, we select the combination that minimizes the used area (although from the standpoint of scheduling only the task-set, the first successful combination would do). The packed schedule of this combination is used to synthesize the contents of the HRT.

3.1.2 Synthesizing HRT contents

There are only five possible processor configurations given 4 superscalar ways: (a) four 1-way partitions, (b) two 1-way partitions and one 2-way partition, (c) two 2-way partitions, (d) one 1-way partition and one 3-way partition, and (e) one 4-way partition. The HRT entries corresponding to each processor configuration (a)-(e) are manually synthesized and shown in Figure 8, respectively. Briefly, for a given processor configuration, a VP is assigned a fraction of the processor's resources equal to its fraction of superscalar ways. For example, in Figure 8(a), each of the 1-way VPs has a 25% share of the processor's resources: each VP owns every function unit (including the fetch unit) for one cycle out of four.

The real-time analysis of Section 3.1.1 may produce a round with multiple processor configurations. For the example in Figure 7, Figure 8(d) is used for the first entry in the HRT (with an LTC of 60 cycles) and Figure 8(b) is used for the second entry in the HRT (with an LTC of 40 cycles and EOT=1).

3.2 Multiple tasks per virtual processor

If there are more tasks in the task-set than the architecture has VPs, then VPs need to support more than just one task each.

The same situation arises in the context of conventional multiprocessors, when there are more tasks than physical processors. In this case, multiple tasks are assigned to each processor, and classical real-time scheduling policies for uniprocessors – *e.g.*, earliest-deadline-first (EDF) or rate-monotonic (RM) [30] – schedule tasks on the same processor. Then, the only question is how to assign tasks to processors, a deeply studied area that is often cast, again, as a bin packing problem with many applicable heuristics [28].

Since VPs are completely decoupled, RVMP can be abstracted as a (flexible) multiprocessor and thus the same techniques apply.

1. Multiple tasks per VP are naturally accommodated by applying (for example) EDF scheduling within VPs. A VP's duty cycle must now accommodate the combined utilization of all tasks assigned to it. That is, the duty cycle of a VP is simply the sum of the duty cycles of tasks assigned to it. Since our procedure of Section 3.1 bin-packs VPs, not tasks, the procedure transparently extends to multiple tasks per VP as long as VPs' duty cycles are calculated using the generalization above.
2. We apply bin packing once again to ensure a good assignment of tasks to VPs, with a subtle enhancement. Since tasks sharing the same VP will ultimately execute on the same number of superscalar ways (the number of ways ultimately allocated to a VP does not vary) an additional consideration when grouping tasks is whether or not they have similar (in-order) instruction-level parallelism (ILP).

4. EXPERIMENTAL METHODOLOGY

The primary experiments involve static worst-case schedulability analysis, which determines the ability to schedule task-sets on various architectures. We compare worst-case schedulability of task-sets on the proposed RVMP architecture and several conventional multiprocessors with equal aggregate resources. Since there is no known static worst-case schedulability analysis framework for SMT (unsafe), it is excluded from the primary experiments.

The primary experiments are followed by secondary experiments, proof-of-concept simulations of the various architectures. Note, dynamic testing does not prove the schedulability of a task-set in the worst-case, only its schedulability for the particular task-set inputs used. Detailed microarchitectural simulation is useful as a prototyping exercise (proof-of-concept) and it also provides a medium for comparing run-time performance of RVMP and SMT for particular task-set inputs.

For the secondary experiments, we use a custom detailed cycle-level simulator that faithfully models the RVMP architecture described in Section 2. The custom simulator was developed using the SimpleScalar toolset [4], insofar as using the SimpleScalar ISA (PISA) and gcc-based compiler. The simulator also models conventional multiprocessor and SMT architectures. Conventional SMT uses out-of-order execution (64-entry reorder buffer), dynamic branch prediction (*gshare* predictor with 2^{16} entries), and hardware-managed caches (the same sizes as RVMP's software-managed scratchpads, borrowed from Ubicom's IP3023 microprocessor [40]: 256KB I-scratchpad and 64KB D-scratchpad). All task-sets are simulated for a complete hyper-period or 100 ms, whichever is less. In these secondary experiments, we compare the run-time performance of (1) RVMP, (2) various equivalent multiprocessor systems with equal aggregate resources, and (3) (unsafe) SMT. Run-time performance is compared in terms of successfully meeting all deadlines or not.

4.1 Static worst-case timing analysis

The real-time scheduling analysis presented in Section 3 requires the WCET for each task, for each of 1-way, 2-way, 3-way, and 4-way partitions. For each task τ, these are referred to as $\text{WCET}_\tau\{1\}$, $\text{WCET}_\tau\{2\}$, $\text{WCET}_\tau\{3\}$, and $\text{WCET}_\tau\{4\}$, respectively. We cannot simply assume that $\text{WCET}_\tau\{4\} = \frac{1}{4} \times \text{WCET}_\tau\{1\}$, because performance does not scale linearly with the number of superscalar ways. Moreover, the worst-case-extended instruction latencies caused by time-sharing contended function units (as explained pre-

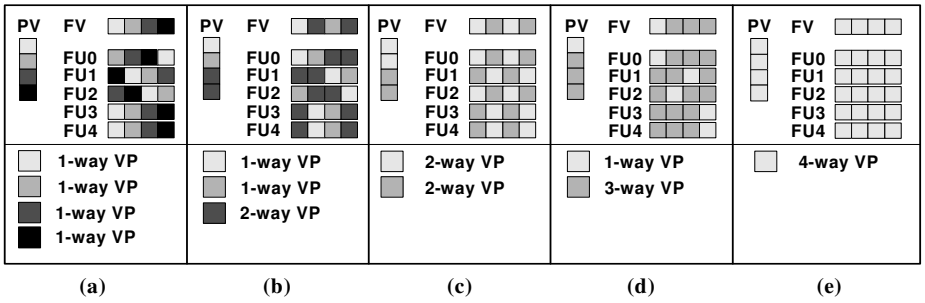


Figure 8: The five possible processor configurations for a 4-way superscalar processor.

viously in Section 2.4) are different among the four cases. WCET analysis needs to be performed specifically for each partition width.

WCET analysis involves identifying the longest timing paths in the program, moving upwards from inner loops and leaf functions towards outer loops and functions at higher levels. Forward branches are handled by selecting the longer of two timing paths, after padding the taken path with the misprediction penalty, since our static branch prediction heuristic predicts forward branches as always not-taken. Backward branches are handled by padding the loop continuation with the misprediction penalty, since our static branch prediction heuristic predicts backward branches as always taken. After identifying longest timing paths, we use simulation to tightly model overlapped execution of instructions along these paths. Note that the scratchpads are partitioned among tasks to eliminate interference and improve analyzability, a common practice in hard-real-time systems [24, 42]. Methods for bounding worst-case memory latency when there are memory accesses from simultaneous tasks [13] (*e.g.*, bus serialization, DRAM bank conflicts, *etc.*) are applicable for RVMP and conventional multiprocessors alike.

4.2 Real-time tasks and task-sets

We use benchmarks from the C-lab real-time benchmark suite [5] and MiBench embedded benchmark suite [16], shown in Table 1. These benchmarks are compiled to the SimpleScalar PISA ISA [4] with -O3 optimization enabled. The first column in Table 1 shows the benchmark names. The second through fifth columns show four WCETs for each task, for each of 1, 2, 3, and 4 ways, respectively.

Table 1: Benchmarks (WCETs in ms at 1GHz).

Task	WCET{1}	WCET{2}	WCET{3}	WCET{4}
cnt	0.118	0.0929	0.0777	0.0777
adpcm	3.06	2.29	1.86	1.64
srt	2.26	1.51	1.13	1.01
mm	2.93	2.29	1.97	1.97
fft	0.692	0.526	0.496	0.447
lms	0.205	0.140	0.123	0.0963
crc	0.0594	0.0513	0.0434	0.0434
toast	0.347	0.261	0.253	0.231
untoast	0.129	0.107	0.0932	0.0925
lame	9.79	7.64	6.95	6.27

Using the tasks above, we generate numerous task-sets with 4 tasks and others with 8 tasks. Tasks are randomly selected for each task-set. The period of every task is randomly selected such that $WCET\{4\} \leq period < 4 \times WCET\{1\}$ (or $8 \times WCET\{1\}$ for task-sets with 8 tasks). The lower bound on the period ensures that any single task will at least be schedulable on a 4-way in-order processor. The upper bound on the period provides some slack for the task-set as a whole to be possibly schedulable.

We define the *scalar utilization* (U_{scalar}) of a task-set as the sum of its tasks' worst-case utilizations according to their WCETs on a scalar in-order pipeline ($\sum_\tau \frac{WCET_\tau\{1\}}{period_\tau}$, for all tasks τ in the task-set). Task-sets are sorted into four different categories (or bins) based on their U_{scalar}. The four bins are as follows: $0 < U_{scalar} \leq 1$, $1 < U_{scalar} \leq 2$, $2 < U_{scalar} \leq 3$, and $3 < U_{scalar} \leq 4$. Each bin has 25 randomly-generated task-sets. These bins represent increasing difficulty in scheduling a task-set, the first bin containing the least demanding task-sets and the fourth bin containing the most demanding task-sets. Any task-set with $U_{scalar} > 4$ is provably unschedulable on all architectures used in the primary experiments.

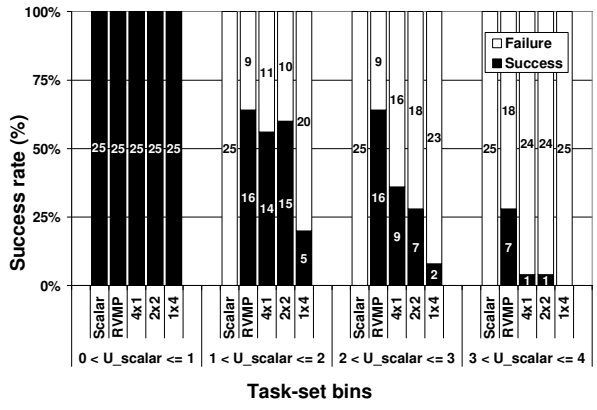
5. RESULTS

5.1 Schedulability tests

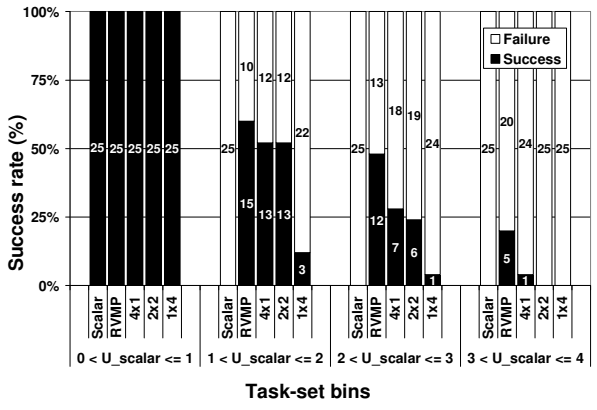
The graph in Figure 9 shows worst-case schedulability results, for various statically analyzable architectures. Figure 9(a) is for task-sets with 4 tasks each and Figure 9(b) is for task-sets with 8 tasks each. For each utilization bin, we plot the number of task-sets in that bin that are schedulable ("Success") versus not schedulable ("Failure") in the worst case, for the various architectures. "Scalar" is the in-order scalar processor, used to calculate the scalar utilization (U_{scalar}) for each of the task-set bins. "RVMP" is our proposed real-time virtual multiprocessor. The other three bars correspond to classic earliest-deadline-first (EDF) scheduling on various conventional uniprocessor and multiprocessor configurations: "4×1" (four in-order scalar processors), "2×2" (two in-order 2-way superscalar processors), and "1×4" (a single in-order 4-way superscalar processor). All architectures have the same frequency (1 GHz). All architectures (except "Scalar") have equal aggregate resources (equal aggregate fetch, issue, and function unit bandwidth). For the conventional multiprocessor systems ("4×1" and "2×2"), the *first-fit-decreasing-utilization* algorithm [34] is used to assign tasks to processors when there are more tasks than processors.

The numbers on the bars represent how many task-sets succeeded and how many failed (out of a total of 25 task-sets per bin). For example, for the second utilization bin ($1 < U_{scalar} \leq 2$) in Figure 9(a), 16 task-sets are schedulable on "RVMP" and 9 task-sets are not, in the worst case.

Task-sets in the first utilization bin are provably schedulable on an in-order scalar processor because their scalar utilizations are less than or equal to 1, therefore we expect these task-sets to be schedulable on all five architectures. This is confirmed in Figure 9(a) and Figure 9(b): all 25 task-sets are schedulable ("Success") on all five architectures. On the other hand, task-sets in the three higher utilization bins ($U_{scalar} > 1$) are provably unschedulable on "Scalar", as confirmed in Figure 9.



(a) 4 tasks per task-set



(b) 8 tasks per task-set

Figure 9: Worst-case schedulability analysis.

As we move from lower to higher utilization bins, scheduling task-sets naturally becomes harder. In all cases, however, "RVMP" successfully schedules more task-sets than all the other architectures, demonstrating greater flexibility compared to rigid multiprocessors. Moreover, flexibility becomes more important for more demanding task-sets. For example, from Figure 9(a), "RVMP" schedules 7 task-sets in the highest utilization bin, whereas the next best architectures ("4×1" and "2×2") schedule only 1 task-set.

The single in-order 4-way superscalar processor, "1×4", successfully schedules considerably fewer task-sets than the other architectures, across the board. This is due to the lack of out-of-order execution of either kind: no OOO execution within tasks (necessary for analyzability) and no OOO execution among tasks ("1×4" is single-threaded).

Figure 9(b) shows that "RVMP" is scalable in terms of supporting more tasks than VPs. For task-sets with 8 tasks, two tasks are scheduled on each VP using EDF scheduling within each VP.

Figure 10 shows a histogram of the various processor configurations used by "RVMP" to schedule the task-sets of Figure 9(a). (For example, "1-3/2-2" denotes two configurations in the round: (i) a 1-way VP and 3-way VP, and (ii) two 2-way VPs.) For task-sets with $U_{scalar} \leq 1$ (not shown here), "RVMP" was configured as four 1-way partitions with no reconfigurations during the round. This is expected due to the low demand of task-sets in that bin (all task-sets were successfully schedulable on "Scalar"). Notice however, that "RVMP" shifts more and more to flexible configurations as the task-sets become more demanding (higher U_{scalar}). This observation is consistent with the results of Figure 9(a). In the highest bin, "RVMP" is still able to schedule 7 demanding task-sets whereas "4×1" and "2×2" only schedule 1 task-set each. Flexible configurations are clearly valuable in this regime.

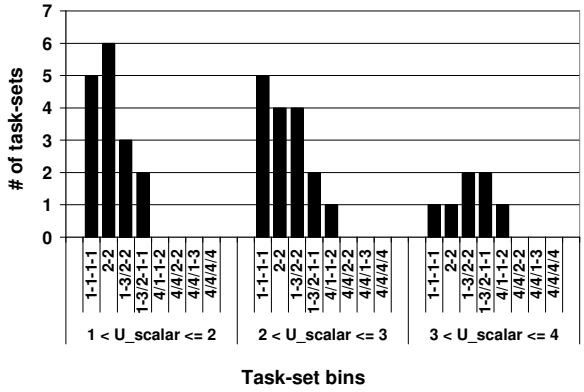


Figure 10: Configuration histogram.

5.2 Run-time experiments

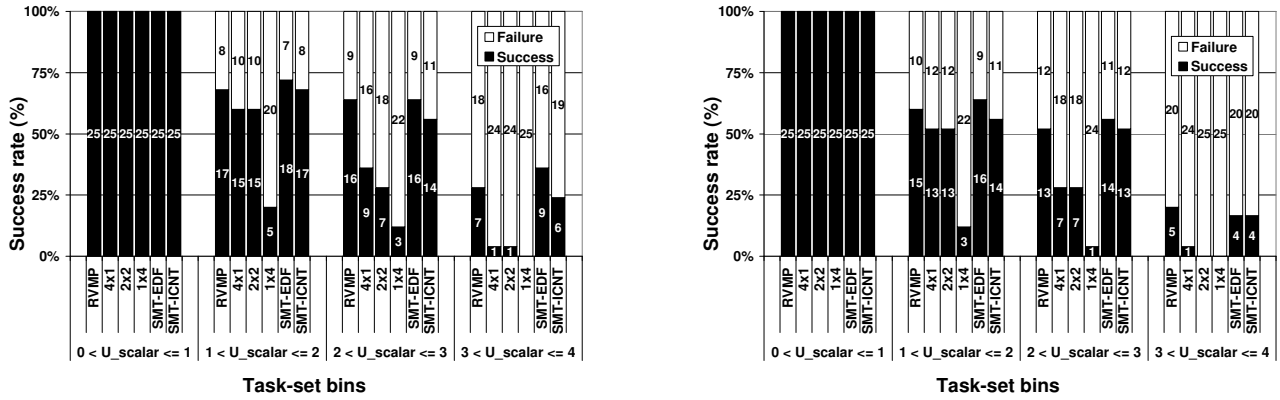
In Figure 11, we show the number of task-sets that succeed or fail at run-time on the various architectures, using our cycle-level simulator. A task-set is considered successful if all deadlines are met for the simulated time-frame (the lesser of the hyper-period or 100 ms). To reiterate, whereas *formal* schedulability results from the previous subsection hold in the *worst case*, regardless of the task-set inputs, simulation results in this section only hold for *specific* task-set inputs.

Two unsafe SMT architectures are now introduced, in addition to the safe architectures used previously for formal schedulability tests: "SMT-EDF" and "SMT-ICNT". "SMT-EDF" uses a real-time-aware (but still not provably safe) thread selection policy that prioritizes tasks according to earliest deadlines [2], while "SMT-ICNT" uses the classic ICOUNT [39] thread selection policy.

The run-time results for the statically analyzable architectures are in agreement with our schedulability tests of the previous subsection. The slight differences between Figure 9 and Figure 11 are due to differences between WCET estimates and actual execution times: actual execution times with specific task-set inputs can naturally be less than WCETs. For example, for the second utilization bin in Figure 9(a), "RVMP" successfully schedules 16 out of the 25 task-sets. However, according to Figure 11(a), 1 additional task-set – only for specific inputs – is successfully scheduled at run-time.

Not only is "RVMP" compatible with hard-real-time system design from the standpoint of a formal schedulability framework, but it also performs comparably to the two dynamic SMT architectures in the run-time comparison. For specific task-set inputs, "SMT-EDF" successfully schedules at run-time no more than two extra task-sets over what "RVMP" schedules. "SMT-ICNT" never successfully schedules more task-sets at run-time than "RVMP" (although they are also close).

These results indicate that, although "RVMP" is more coarse-grain in its space/time partitioning than SMT, the partitioning is flexible enough in practice to match SMT, thus successfully combining both analyzability and high performance. With dynamic SMT, on the other hand, there is no way to tell *a priori* which task-



(a) 4 tasks per task-set

(b) 8 tasks per task-set

Figure 11: Run-time experiments.

sets will succeed in the worst case. As such, it is unsafe to rely on dynamic SMT in hard-real-time systems. A closer comparison of "SMT-EDF" and "SMT-ICNT" provides run-time evidence of this safety issue. We find that among 15 unique task-sets scheduled by either "SMT-EDF" or "SMT-ICNT" for the third utilization bin in Figure 11(b), 12 task-sets are scheduled by both, 2 are scheduled by only "SMT-EDF", and 1 is scheduled by only "SMT-ICNT". The latter 3 task-sets demonstrate that dynamic thread selection affects schedulability, which then raises the deeper concern that any subtle interference can throw things off.

6. RELATED WORK

Contemporary static worst-case timing analysis tools can derive tight and safe WCETs of tasks running on in-order scalar [18, 27] and in-order superscalar [29, 32] pipelines. Recent attempts to analyze out-of-order (OOO) scalar pipelines [26] are so far limited by simplifying assumptions. Nonetheless, future techniques for analytically bounding OOO execution can certainly be exploited within our RVMP framework, since RVMP guarantees non-interference among in-order and OOO VPs alike, a key underlying assumption of WCET analysis (tasks are analyzed individually).

Recent work explores switch-on-event multithreading (to hide memory latency) in real-time systems [25, 9, 13]. Kreuzinger *et al.* [25] do not provide an analytical framework for provably bounding the amount of overlap among tasks, rather they only perform dynamic testing. Crowley and Baer [9] derive the combined WCET of overlapped tasks. Their technique must consider all possible overlap scenarios among tasks and also it is not compatible with arbitrary scheduling policies. El-Haj-Mahmoud and Rotenberg [13] analytically bound computation/memory overlap among tasks yielding a closed-form schedulability test. The latter two analytical works [9, 13], while safe in terms of worst-case analysis, are limited to scalar pipelines with only one of the hardware threads selected for execution on the pipeline at a time. This paper provides an analytical approach to multithreaded superscalar processors, a significant performance leap.

Several papers [*e.g.*,36,38,14] evaluate policies to share resources among threads in SMT processors, to address both throughput and fairness. Cazorla *et al.* [6] provision SMT for quality-of-service (QoS). However, no hard guarantees can be made regarding the performance of threads, because of the dynamically-scheduled processor underneath and also because no analytical framework is attempted. Likewise, SMT resource sharing policies proposed by Jain *et al.* [22] are applicable to soft-real-time systems only (schedulability is evaluated via dynamic testing, and a task-set is considered schedulable even if 5% of deadlines are missed).

Static resource partitioning has been previously proposed in the context of VLIW architectures, such as XIMD [43]. Replicated sequencers and homogeneous function units give XIMD true multiprocessor qualities, albeit with the ability to gang together FUs to form different-width execution backends. Our work makes it possible to carve out arbitrary interference-free partitions in the context of a contemporary superscalar processor with one shared fetch unit and heterogeneous function units. And, we develop a novel real-time scheduling formalism to go with the architecture.

Weld [35] dynamically combines instructions from a main thread and a future speculative thread into one VLIW word. A similar dynamic approach for combining instructions from different threads into a single VLIW word is used by the M-Machine [15] and others [23, 21]. Again, dynamic approaches are not suitable for hard-real-time systems as there is no corresponding analytical framework.

Software thread integration (STI) [10] combines instructions from dual threads (a non-real-time guest thread plus a real-time host thread) into one binary. Integrating threads is complicated by incompatible control-flow, limiting integration opportunities. Our decoupled architecture works with arbitrary task-sets and does not require combined task compilation and analysis.

The Ubicom IP3023 microprocessor [40] was designed with analyzable high performance in mind. The IP3023 provides 8 hardware threads that share a 10-stage in-order scalar pipeline with static branch prediction and I- and D-scratchpad memories. A 64-entry cyclic table, the hard-real-time table or HRT (a term we borrow in this paper), specifies which thread to fetch from next, on a cycle-by-cycle basis. Cycling through threads has certain elements of the HEP machine [37]. The IP3023 is also quite similar to the scalar DISC architecture [33], which assigns a statically-guaranteed percentage of processor cycles to *execution streams*. The IP3023 (and precursors, HEP and DISC) is scalar and as such is not concerned with providing interference-free different-width partitions based on aggregating ways of a superscalar substrate. Our novel RVMP architecture and real-time scheduling framework take Ubicom's processor strategy to a whole new level.

7. SUMMARY

We addressed the trade-off in real-time architectures between high performance and analyzability. We proposed the novel Real-time Virtual Multiprocessor (RVMP) architecture and matching real-time scheduling analysis. RVMP virtualizes a single in-order super-

scalar processor into multiple interference-free different-sized virtual processors, in both space and time. We proposed a simple real-time scheduling approach that concentrates scheduling into a small time interval, proving or disproving schedulability with very low complexity and at the same time producing a compact space/time schedule that orchestrates virtualization. RVMP successfully combines the analyzability (hence real-time formalism) of multiple processors with the flexibility (hence high performance) of simultaneous multithreading (SMT). Worst-case schedulability experiments show that RVMP is able to schedule more task-sets than rigid multiprocessor counterparts. We also observed that RVMP's advantage increases as task-sets become more demanding. Moreover, RVMP is as effective as an SMT processor in run-time experiments, meanwhile providing a real-time formalism that SMT does not currently provide.

8. REFERENCES

[1] A. Anantaraman, K. Seth, K. Patil, E. Rotenberg, and F. Mueller. Virtual Simple Architecture (VISA): Exceeding the Complexity Limit in Safe Real-Time Systems. In *Proc. of the 30th Int'l Symp. on Computer Architecture*, pp. 350–361, June 2003.

[2] A. Anantaraman, K. Seth, E. Rotenberg, and F. Mueller. Exploiting VISA for Higher Concurrency in Safe Real-Time Systems. Tech. Report TR-2004-15, CS Dept., NC State Univ., May 2004.

[3] F. Bodin and I. Puaut. A WCET-Oriented Static Branch Prediction Scheme for Real-Time Systems. In *Proc. of the 17th Euromicro Conf. on Real-Time Systems*, July 2005.

[4] D. Burger, T. M. Austin, and S. Bennett. The Simplescalar Toolset, Version 2. Tech. Report CS-TR-1997-1342, CS Dept., Univ. of Wisconsin-Madison, July 1997.

[5] C-Lab. WCET Benchmarks. Available from `http://www.c-lab.de/index.php?id=462&L=3`.

[6] F. J. Cazorla, A. Ramírez, M. Valero, P. M. W. Knijnenburg, R. Sakellariou, and E. Fernández. QoS for High-Performance SMT Processors in Embedded Systems. *IEEE Micro*, 24(4):24–31, 2004.

[7] B. Chazelle. The Bottom-Left Bin-Packing Heuristic: An Efficient Implementation. *IEEE Trans. on Computers*, 32(8):697–707, 1983.

[8] D. Cormie. The ARM11 Microarchitecture. White paper, Apr. 2002.

[9] P. Crowley and J.-L. Baer. Worst-Case Execution Time Estimation of Hardware-assisted Multithreaded Processors. In *Proc. of the 2nd Workshop on Network Processors*, pp. 36–47, Feb. 2003.

[10] A. G. Dean and J. P. Shen. Techniques for Software Thread Integration in Real-Time Embedded Systems. In *Proc. of the 19th Int'l Real-Time Systems Symp.*, pp. 322–333, Dec. 1998.

[11] J. E. G. Coffman, M. R. Garey, and D. S. Johnson. Approximation Algorithms for Bin Packing: A Survey. In *Approximation Algorithms for $\mathcal{NP}$-Hard Problems*, pp. 46–93. PWS Publishing Co., 1997.

[12] J. H. Edmondson, P. Rubinfeld, R. Preston, and V. Rajagopalan. Superscalar Instruction Execution in the 21164 Alpha Microprocessor. *IEEE Micro*, 15(2):33–43, 1995.

[13] A. El-Haj-Mahmoud and E. Rotenberg. Safely Exploiting Multithreaded Processors to Tolerate Memory Latency in Real-Time Systems. In *Proc. of the 2004 Int'l Conf. on Compilers, Architecture, and Synthesis for Embedded Systems*, pp. 2–13, Sep. 2004.

[14] A. El-Moursy and D. H. Albonesi. Front-End Policies for Improved Issue Efficiency in SMT Processors. In *Proc. of the 9th Int'l Symp. on High Performance Computer Architecture*, pp. 31–42, Feb. 2003.

[15] M. Fillo, S. W. Keckler, W. J. Dally, N. P. Carter, A. Chang, Y. Gurevich, and W. S. Lee. The M-Machine Multicomputer. In *Proc. of the 28th Int'l Symp. on Microarchitecture*, pp. 146–156, Nov. 1995.

[16] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. MiBench: A Free, Commercially Representative Embedded Benchmark Suite. In *Proc. of the 4th Workshop on Workload Characterization*, Dec. 2001.

[17] T. Hand. Real-Time Systems Need Predictability. *Computer Design (RISC Supplement)*, pp. 57–59, Aug. 1989.

[18] M. G. Harmon, T. P. Baker, and D. B. Whalley. A Retargetable Technique for Predicting Execution Time of Code Segments. In *Proc. of the 13th Int'l Real-Time Systems Symp.*, pp. 68–77, Dec. 1992.

[19] S. Hily and A. Seznec. Out-of-Order Execution may not be Cost-Effective on Processors Featuring Simultaneous Multithreading. In *Proc. of the 5th Int'l Conf. on High Performance Computer Architecture*, p. 64, Jan. 1999.

[20] IBM Corp. IBM PowerPC 740 / PowerPC 750 RISC Microprocessor User's Manual. Feb. 1999.

[21] B. Iyer, S. Srinivasan, and B. L. Jacob. Extended Split-Issue: Enabling Flexibility in the Hardware Implementation of NUAL VLIW DSPs. In *Proc. of the 31st Int'l Symp. on Computer Architecture*, pp. 364–375, June 2004.

[22] R. Jain, C. J. Hughes, and S. V. Adve. Soft Real-Time Scheduling on Simultaneous Multithreaded Processors. In *Proc. of the 23rd Int'l Real-Time Systems Symp.*, pp. 134–145, Dec. 2002.

[23] S. Kaxiras, G. J. Narlikar, A. D. Berenbaum, and Z. Hu. Comparing Power Consumption of an SMT and a CMP DSP for Mobile Phone Workloads. In *Proc. of the 2001 Int'l Conf. on Compilers, Arch., and Syn. for Embedded Systems*, pp. 211–220, Nov. 2001.

[24] D. B. Kirk. SMART (Strategic Memory Allocation for Real-Time) Cache Design. In *Proc. of the 10th Int'l Real-Time Systems Symp.*, pp. 229–239, Dec. 1989.

[25] J. Kreuzinger, A. Schulz, M. Pfeffer, and T. Ungerer. Real-Time Scheduling on Multithreaded Processors. In *Proc. of the 7th Workshop on Real-Time Computing and Applications Symp.*, pp. 155–159, Dec. 2000.

[26] X. Li, A. Roychoudhury, and T. Mitra. Modeling Out-of-Order Processors for Software Timing Analysis. In *Proc. of the 25th Int'l Real-Time Systems Symp.*, pp. 92–103, Dec. 2004.

[27] Y.-T. S. Li, S. Malik, and A. Wolfe. Efficient Microarchitecture Modeling and Path Analysis for Real-Time Software. In *Proc. of the 16th Int'l Real-Time Systems Symp.*, pp. 298–307, Dec. 1995.

[28] J. Liebeherr, A. Burchard, Y. Oh, and S. H. Son. New Strategies for Assigning Real-Time Tasks to Multiprocessor Systems. *IEEE Trans. on Computers*, 44(12):1429–1442, 1995.

[29] S.-S. Lim, J. H. Han, J. Kim, and S. L. Min. A Worst Case Timing Analysis Technique for Multiple-Issue Machines. In *Proc. of the 19th Int'l Real-Time Systems Symp.*, pp. 334–345, Dec. 1998.

[30] C. L. Liu and J. W. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *Journal of the ACM*, 20(1):46–61, 1973.

[31] J. Liu. *Real-Time Systems*. Prentice Hall, 2000.

[32] T. Lundqvist and P. Stenström. An Integrated Path and Timing Analysis Method based on Cycle-Level Symbolic Execution. *Real-Time Systems*, 17(2-3):183–207, 1999.

[33] M. D. Nemirovsky, F. Brewer, and R. C. Wood. DISC: Dynamic Instruction Stream Computer. In *Proc. of the 24th Int'l Symp. on Microarchitecture*, pp. 163–171, Nov. 1991.

[34] Y. Oh and S. H. Son. Fixed Priority Scheduling of Periodic Tasks on Multiprocessor Systems. CS Tech. Report, Univ. of Virginia, 1995.

[35] E. Özer, T. M. Conte, and S. Sharma. Weld: A Multithreading Technique Towards Latency-Tolerant VLIW Processors. In *Proc. of the 8th Int'l Conf. on High Perf. Comp.*, pp. 192–203, Dec. 2001.

[36] S. Raasch and S. Reinhardt. The Impact of Resource Partitioning on SMT Processors. In *Proc. of the 12th Int'l Conf. on Parallel Architectures and Compilation Techniques*, pp. 15–26, Sep. 2003.

[37] B. Smith. Architecture and Applications of the HEP Multiprocessor Computer System. In *Proc. of the 4th Symp. on Real Time Signal Processing IV*, pp. 241–248, 1981.

[38] N. Tuck and D. M. Tullsen. Initial Observations of the Simultaneous Multithreading Pentium 4 Processor. In *Proc. of the 12th Int'l Conf. on Parallel Arch. and Compilation Tech.*, pp. 26–35, Sep. 2003.

[39] D. Tullsen, S. Eggers, J. Emer, H. Levy, J. Lo, and R. Stamm. Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor. In *Proc. of the 23rd Int'l Symp. on Computer Architecture*, pp. 191–202, May 1996.

[40] Ubicom, Inc. The Ubicom IP3023 Wireless Network Processor. White paper, Apr. 2003.

[41] S. Udayakumaran and R. Barua. Compiler-Decided Dynamic Memory Allocation for Scratch-pad Based Embedded Systems. In *Proc. of the 2003 Int'l Conf. on Compilers, Architecture and Synthesis for Embedded Systems*, pp. 276–286, Sep. 2003.

[42] A. Wolfe. Software-Based Cache Partitioning for Real-Time Applications. *Computer Software Engineering*, 2(3):315–327, 1994.

[43] A. Wolfe and J. P. Shen. A Variable Instruction Stream Extension to the VLIW Architecture. In *Proc. of the 4th Int'l Conf. on Arch. Support for Prog. Lang. and Operating Systems*, pp. 2–14, Apr. 1991.