Architectural Support
for Real-Time Task Scheduling
in SMT Processors

Francisco J. Cazorla
Peter M.W. Knijnenburg
Rizos Sakellariou
Enrique Fernandez
Alex Ramirez
Mateo Valero
Motivation

• In Simultaneous Multithreaded processors (SMTs), the front end of a superscalar processor is adapted in order to fetch instructions from several threads, while the back end is shared between these threads.

• This gives a good performance/cost ratio.

• Many current and future processors have SMT capabilities (P4, Power5).

• There also exist embedded SMTs (Meta).
Motivation

• A fetch policy decides from which threads to fetch instructions.

• A well known fetch policy is icount that fetches from threads with the fewest instructions in the front end.

• However, execution times of applications are highly dependent on the other threads in the workload.

• This poses problems if SMTs are to be used in real time environments.

• This presentation discusses two extensions of SMTs to deal with this problem.
Existing Approaches

• Assume that one thread in a workload is a real time thread.

• Jain et al. and Snavely et al. propose to profile all possible combinations of applications to find a workload that has highest *symbiosis*.

• Next, they adapt *icount* with handicap numbers to prioritize the fetch from the real time thread.

• The main drawback is the high number of profiles needed and the real time thread misses its deadline often.
• Cazorla et al. extend the hardware to run a given thread at a given percentage of its full speed, called *Predictable Performance*.

• The main drawback is that this mechanism is very expensive.
Approach

• We *partition* the hardware resources between the critical and the non-critical thread and to reserve a *minimum* fraction of the resources for the CT that enables it to meet its deadline.

• This implies that the NCT achieves high performance as well.

• The SMT is extended with a *Resource Allocator* that assigns resources to threads and tracks the amount of resources used.
• The OS level job scheduler is extended slightly to provide the Resource Allocator with some information on how to partition resources.
Resources vs. Performance

![Graph showing the relationship between resources and relative IPC for different applications. The graph includes lines for g721_d, gsm_c, gsm_d, mpeg2_c, and mpeg2_d.](image)
OS Job Scheduler

- Our proposed method exploits the relation between the amount of resources given to the critical thread and the performance it obtains.

- When the OS level job scheduler wants to execute a critical thread, given its $WCET$ and a period $P$, it simply computes the allowable performance slow down, $S$, given by $S = \frac{P}{WCET}$.

- For such a value of $S$, each instance of this job finishes before its deadline.

- Suppose the real execution time of an instance $i$ is $T_i$. Then $S \cdot T_i = \frac{P}{WCET} \cdot T_i \leq \frac{P}{WCET} \cdot WCET = P$. 
Questions

• We need to determine which resources are being controlled by the resource allocator.

• We need to decide whether the job scheduler or the resource allocator determines the exact amount of resources given to the critical thread.
  
  – A resource allocation is fixed for the entire period by the job scheduler: the *static approach*.
  
  – The resource allocator can dynamically vary the amount of resources dedicated to the critical thread: the *dynamic approach*. 
Resources

- Shared resources: fetch bandwidth, issue queues, issue bandwidth, physical registers, instruction cache, L1 data cache, unified L2 cache, and TLBs.

- We have shown that for media applications, cache miss rates are very small and we do not need to control them.

- Issue bandwidth also leads to very small differences in performance. Hence, we do not control this either.

- Controlling the other resources gives good results.
Variability in IPC of Critical Thread for 8 Workloads
Static Approach

• The job scheduler computes *a priori* the resource partitioning that is used throughout the entire period of the critical thread.

• Given the slow down factor $S = \frac{P}{WCET}$, we need to compute a function $f(S) = Y$ so that the critical thread receives $Y\%$ of the resources in order to get $S\%$ of maximum performance.

• Function is called *performance/resource function* or *p/r function*. 
Profile-Directed P/R Function

The graph shows the relationship between the relative IPC (%) on the x-axis and the amount of resources (%) on the y-axis. The lines represent different profiles:
- `adpcm_c real`
- `adpcm_c adhoc`
- `gsm_d real`
- `gsm_d adhoc`

The data points indicate how the amount of resources changes as the relative IPC increases for each profile.
Dynamic Approach

- The resource allocator dynamically determines the amount of resources given to the critical thread.

- The main advantage is that it adapts to program execution phases.

- Based on the observation that in order to realize $X\%$ of the overall IPC for a given job, it is sufficient to realize $X\%$ of the maximum local IPC at every instant throughout its execution.

- The dynamic approach is a simplification of *Predictable Performance* (CF2004).
Dynamic Approach: Schematic View

CT sample phase
60K cycles

CT tune phase
1.2M cycles

CT sample phase
60K cycles

CT full speed
determination

resource allocation
re-adjustment

...
Resource Allocator

• The resource allocator distinguishes two phases that are executed in alternate fashion.

• During the first phase, the sample phase, all resources under control are given to the CT and we obtain an estimate of the current $IPC_{alone}$ of the CT.

• During the second phase, the tune phase, our mechanism dynamically varies the amount of resources given to the CT to achieve an IPC that is equal to the local $IPC_{alone} \times S$. 
Sample Phase

- The sample phase takes 60,000 cycles.

- Divided in two sub-phases
  - Warmup sub-phase of 50,000 cycles to remove interference from the shared resources
  - Actual sample sub-phase of 10,000 cycles to measure the local $IPC_{alone}$ of the CT.
Tune Phase

• The tune phase takes 300,000 cycles.

• Divided in sub-phases of 15,000 cycles in which the realized \( IPC \) of the CT is measured.

• If this \( IPC \) is lower than \( S \times IPC_{alone} \), more resources are given to the critical thread.

• Otherwise, more resources are given to the non-critical thread.
Experiment: Benchmarks

- All MediaBench benchmarks as critical threads.

- SPEC2000 applications as non-critical threads.

- Total of 80 workloads of two threads.

- We consider low, medium, and high utilizations.
SMT Simulator

Trace driven SMT simulator based on *smtsim* using an aggressive configuration.

<table>
<thead>
<tr>
<th>Processor Configuration</th>
<th>12 stages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pipeline depth</td>
<td>8</td>
</tr>
<tr>
<td>Fetch/Issue/Commit Width</td>
<td>64 int, 64 fp, 64 ld/st</td>
</tr>
<tr>
<td>Queues Entries</td>
<td>6 int, 3 fp, 4 ld/st</td>
</tr>
<tr>
<td>Execution Units</td>
<td>256 integer, 256 fp</td>
</tr>
<tr>
<td>Physical Registers</td>
<td>512 entries</td>
</tr>
<tr>
<td>(shared)ROB size</td>
<td>16K entries gshare</td>
</tr>
<tr>
<td>Branch Predictor</td>
<td>256-entry, 4-way assoc.</td>
</tr>
<tr>
<td>Branch Target Buffer</td>
<td>256 entries</td>
</tr>
<tr>
<td>Return Address Stack</td>
<td></td>
</tr>
<tr>
<td>Icache, Dcache</td>
<td>64 Kbytes, 2-way, 8-bank, 64-byte lines, 1 cycle access</td>
</tr>
<tr>
<td>L2 cache</td>
<td>2048 Kbytes, 8-way, 8-bank, 64-byte lines, 20 cycle access</td>
</tr>
<tr>
<td>Main memory latency</td>
<td>300 cycles</td>
</tr>
<tr>
<td>TLB miss penalty</td>
<td>160 cycles</td>
</tr>
</tbody>
</table>
Results for Static Approach
Success Rate

![Graph showing success rate variations with respect to deadline.](image)
Results for Static Approach
Throughput

![Bar Chart]

- Linear
- Power 0.7
- Power 0.4
- Adhoc

NCT Performance

- High
- Medium
- Low
Results for All Approaches
Success Rate

- Fetch control
- Resource Allocator (static)
- Resource Allocator (dynamic)
- Predictable Performance
- Mean5WorseVariance

Success Rate

variation with respect to deadline (%)
Results for All Approaches

Throughput

- Fetch control
- Resource Allocator (static)
- Resource Allocator (dynamic)
- Predictable Performance
Costs

- Fetch control only requires handicap numbers.

- Static approach uses Resource Allocator which contains some counters and logic.

- Dynamic approach moreover needs logic for sample and tune phases.

- Predictable performance needs more logic to monitor and partition the L2 cache.
Applicability

- Fetch control and static approach can be used for all applications.

- Dynamic approach requires that the non-critical thread does not interfere too much with the critical thread in the L2 cache.

- Predictable Performance requires that all instances of an application have more or less the same IPC.
## Trade-Off

<table>
<thead>
<tr>
<th>Metric</th>
<th>fetch control</th>
<th>static</th>
<th>dynamic</th>
<th>PP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Success rate</td>
<td>-- (&lt;75%)</td>
<td>++ (&gt;95%)</td>
<td>++</td>
<td>++</td>
</tr>
<tr>
<td>Throughput NCT</td>
<td>--</td>
<td>+</td>
<td>++</td>
<td>++</td>
</tr>
<tr>
<td>Applicability</td>
<td>++</td>
<td>++</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>Cost</td>
<td>--</td>
<td>-</td>
<td>+</td>
<td>++</td>
</tr>
</tbody>
</table>
Implementation Resource Allocator

- PCs → Fetch → Instruction Cache → Decode 1 → Decode 2 → Rename → Instr. Queues → ...
- CT's limit registers
- NCT's limit registers
- Updated by the OS
- Instruction is committed

(1) Decode 1
(2) Instr. Queues
(3) CT's usage counters
(4) OR
(5) OR
Conclusions

• We have proposed two novel approaches to the problem of enabling SMT processors for soft-real time systems.

• Our methods are based on resource partitioning, reserving a minimum fraction of all resources for the critical thread.

• Our methods do not require any knowledge beyond information that is traditionally used by the OS level job scheduler.

• We significantly outperform fetch control and are almost as good as Predictable Performance using a much less complex mechanism.
• On average, the critical thread meets its deadline in 98% of the cases considered.