Exploiting Subtrace-Level Parallelism in Clustered Processors

Rafael Ubal, Julio Sahuquillo, Salvador Petit, Pedro López, and Jose Duato
Dept. of Computing Engineering (DISCA)
Universidad Politécnica de Valencia
Camino de Vera, s/n, 46021, Valencia, Spain
raurte@gap.upv.es

Categories and Subject Descriptors: B.8.2 [Hardware]: Performance and Reliability—Performance Analysis and Design Aids; C.1 [Computer Systems Organization]: Processor Architectures

General Terms: Performance.

Keywords: Clustered processors, subtraces, trace cache, computer architecture.

1. MOTIVATION

Multicore chips are currently dominating the microprocessor market as designs that improve performance and sustain power consumption. However, complex core features must be still considered to provide good performance for existing sequential applications. An effective approach to reduce core complexity without dramatically sacrificing performance is to distribute critical processor structures by using clustered microarchitectures. In these designs, the inter-cluster communication latency has a critical impact on global performance [1], and thus, keeping the number of copy instructions as low as possible becomes a major design concern. While sophisticated steering algorithms have been designed for this aim, this bottleneck can still be further reduced.

To this end, our proposal first aims at dynamically generating independent chains of instructions (subtraces) out of traces of sequential code, which are then steered to different clusters. Subtraces are generated by analyzing and splitting a sequence of committed instructions. Then, individual instructions are replicated in several subtraces, until they become completely independent from each other. Hereby, additional parallelism is artificially induced as long as it helps further alleviate the inter-cluster communication bottleneck. The proposed mechanism has been evaluated on top of a clustered trace-cache x86 microprocessor model, where the trace cache fill unit has been tailored to detect and construct independent subtraces, after the commit stage and out of the critical path. This information is then reused by the steering logic, which may insert instruction replicas into several clusters.

A block diagram of the baseline clustered microarchitecture is shown in Figure 1a, where a front-end provides instructions to a shared ROB and LSQ, which are steered into several clusters communicated through an interconnection network. Figure 1b shows the structure of one cluster with a private IQ, register file, and functional unit pool (Figure 1b). Finally, Figure 1c shows the scheme of the processor front-end with a trace cache, whose trace lines are filled with traces of committed instructions built at the fill unit.

Figure 1: Clustered microarchitecture block diagram.

2. GENERATION OF SUBTRACES

Given an original sequence of instructions forming a trace, subtrace-level parallelism is obtained by decomposing it into two or more independent subtraces that may have instructions in common. Figure 2a shows a portion of code as a directed graph, where each instruction is represented by a vertex, and each dependence among instructions is represented by an arc. Figures 2b and 2c show two subgraphs, each corresponding to a subtrace, whose superposition contains all arcs and vertexes of the original graph. Subtraces are generated in such a way that all input dependences are satisfied for each instruction, by probably executing some instructions in both subtraces. When independent subtraces are wisely steered to different clusters, inter-cluster communication can be reduced at the expense of some replicated work, which usually entails global performance benefits.

The proposed algorithm consists of two phases implemented in the fill unit, which is placed after the commit stage in a superscalar processor pipeline, and out of the critical path. After processing a trace $Tr$ of $N$ committed instructions, it is split into $c$ independent subtraces ($St_0$, $St_1$, etc.) of maximum size $N$, where $c$ is the number of clusters. Figure 3 represents an example using two clusters and the flow of instructions shown in Figure 2.

Phase 1. Selection of subtrace. As shown in Figure 3a, instructions from trace $Tr$ are first split individually among subtraces $St_0$

Copyright is held by the author/owner(s).
PACT '10, September 11–15, 2010, Vienna, Austria.
ACM 978-1-4503-0178-7/10/09.
and $STr_1$. When an uop commits, it is assigned to that subtrace containing its input operands. If input operands are split among subtraces, or there is no subtrace containing them, then the least loaded subtrace is chosen. Likewise, if a given imbalance among subtraces is reached, the least loaded subtrace is chosen and the presence of input operands is ignored. The subtrace imbalance counter is based on subtrace length deviations, and the optimal subtrace imbalance threshold has been experimentally fixed to 8.

Phase 2. Satisfying dependencies. After committing $N$ instructions, subtraces are processed so as to make them independent from each other. To this end, the input dependences of each instruction are satisfied by inserting their producers into the same subtrace, which might cause producer instructions to be replicated in several subtraces. As shown in Figure 3b, subtraces are processed in this phase from left to right, i.e., from the youngest to the oldest instruction. Thus, each new instruction placed in subtrace $STr_x$ will cause all its older producers to be inserted into $STr_x$ as well. In the example, the processing of instruction $f$ makes $a$ be inserted into $STr_1$, and the processing of $c$ makes its producer $b$ be inserted into $STr_0$.

3. PERFORMANCE RESULTS

The performance evaluation has been carried out on top of the Multi2Sim 2.2 simulation framework [2], a cycle-accurate simulator for x86-based superscalar processors, extended to model a clustered architecture with support for independent subtraces generation. The parameters of the modeled machine are summarized in Table 1. The Mediabench suite has been used to stress the machine, and simulations are stopped after the first 100 million uops commit. The steering algorithm and the interconnection network among clusters are important design factors related with the criticality of the inter-cluster communication latency. For a good baseline performance, the modeled schemes use a sophisticated steering algorithm called topology-aware steering [3], and several interconnection networks with different realistic link delays are considered.

Performance speedups have been computed for a clustered processor generating independent subtraces with respect to the baseline machine of the same characteristics. The number of clusters has been ranged from 2 to 8, and the evaluated inter-cluster networks include a bus with 2, 4, and 8 cycles link delays, a crossbar (or $n$-buses) with 2, 4, and 8 cycles link delays, and a mesh with 2- and 4-cycle links. The speedup values summarized next correspond to the average values computed for the whole benchmark suite (16 workloads).

- In a bus topology, the fastest configuration (2-cycle links) provides speedups below 5%. However, a fast bus is only suitable for a very low number of connected clusters. When the link delay is increased to 4, a 4-cluster configuration provides a speedup greater than 10%, while an 8-cycle latency value makes subtraces outperform the baseline machine by more than 15% for some configurations. When the communication latency increases, a reduction of copy instructions has a stronger benefit on performance.
- Crossbar topologies eliminate packet collisions when different pairs of clusters communicate, which improves performance of both the baseline and proposed designs. Since copy instructions do not incur such a high penalty, speedups decrease. However, the hardware cost of a crossbar grows quadratically with the number of clusters. Thus, this topology might be unfeasible for a high number of clusters.
- Finally, a mesh can afford shorter point-to-point link delays. The main communication latency occurs in this case when distant clusters communicate and need to traverse several routers. A mesh with 4-cycle link delays provides speedups above 5%, regardless of the number of clusters.

Table 1: Baseline machine parameters.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decode, dispatch, steer, commit bandwidth</td>
<td>8 uops/cycle</td>
</tr>
<tr>
<td>Issue width</td>
<td>2 uops/cycle (each cluster)</td>
</tr>
<tr>
<td>Trace cache</td>
<td>256 traces (64 sets, 4 ways), 16-uop traces</td>
</tr>
<tr>
<td>Global storage resources</td>
<td>256-entry ROB, 64-entry LSQ</td>
</tr>
<tr>
<td>Private resources per cluster</td>
<td>40-entry IQ, 92-entry RF</td>
</tr>
<tr>
<td>Functional units per cluster and latency (total/issue)</td>
<td>4 Int. add (2/1), 1 Int. mult. (5/5), 1 Int. div (2/10)</td>
</tr>
<tr>
<td>Branch predictor type</td>
<td>Hybrid (2-level + bimodal)</td>
</tr>
<tr>
<td>Memory Hierarchy</td>
<td></td>
</tr>
<tr>
<td>L1 cache</td>
<td>32 KB, 2-way, 64-byte block, 2-cycle latency</td>
</tr>
<tr>
<td>L2 cache</td>
<td>512 KB, 8-way, 64-byte block, 10-cycle latency</td>
</tr>
<tr>
<td>Main memory</td>
<td>100-cycle access time</td>
</tr>
</tbody>
</table>

4. REFERENCES