Hardware-Based Generation of Independent Subtraces of Instructions in Clustered Processors

R. Ubal
Electrical and Computer Engineering Dept.
Northeastern University
Boston (MA), USA
J. Sahuquillo, S. Petit, P. López, and J. Duato
Dept. of Computer Engineering (DISCA)
Universidad Politécnica de Valencia
Valencia, Spain

Abstract—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, communication latency among clusters is a critical performance bottleneck, and a good steering algorithm is required to reduce inter-cluster communication.
In this paper, we propose a new energy-efficient microarchitectural approach that reduces inter-cluster communication by detecting and generating independent chains of instructions, referred to as subtraces, from the execution of sequential programs. The devised mechanism has been modeled on an x86-based trace-cache processor, where subtraces are built in the fill unit, stored in a trace cache, and individually steered to different clusters. Experimental results show that the proposal reaches performance speedups around 7% and 15% for point-to-point and bus-based interconnects, respectively, while achieving energy savings of up to 12%.

Index Terms—Clustered Processors, Subtraces, Parallelism.

1 Introduction

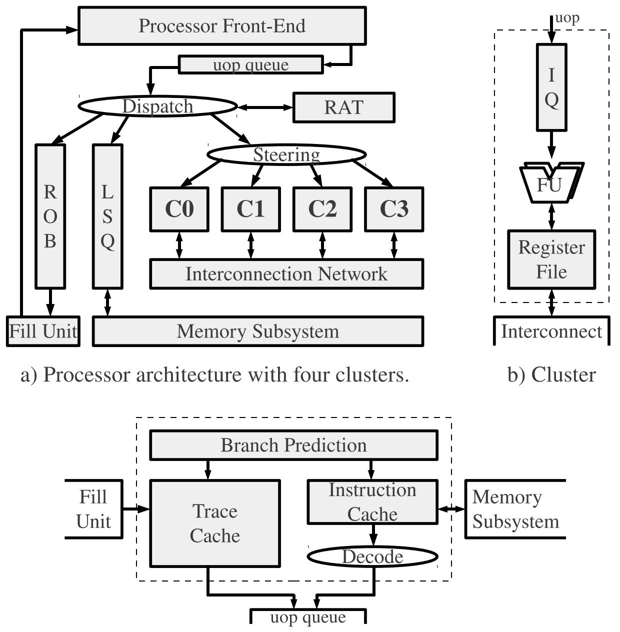
During nearly the last two decades, performance of superscalar microprocessors has been improved by exploiting ILP through increasingly more aggressive mechanisms that maintain binary compatibility. However, continuous shrinking of the transistor size causes an increase of power density, while performance does not rise at the same pace. Microprocessor industry has moved to multicore processors in order to trade off power consumption and global performance. In these processors, the number of cores, as well as their individual complexity, widely differs among industry products, which range from simple in-order execution cores [1] to complex out-of-order execution processors implementing simultaneous multithreading [2].

Though most recent multicore-related research focuses on a large number of cores, industry is still providing products with a rather low number of cores. The main reason for this situation is the limited software and operating system scalability, as well as the fact that most current applications are designed with traditional sequential programming techniques.

This work focuses on a new microarchitectural approach to extract parallelism at run time from sequential applications. To this end, we concentrate on clustered microarchitectures, which were proposed to reduce the complexity of non-scalable structures in superscalar processors [3]. In these architectures, each cluster contains its own instruction queue, register file, and functional units, whereas a common processor front-end (fetch, decode, and renaming logic) is shared among them. Since global complexity is reduced, a clustered architecture is suitable both for monolithic and multicore processors.

After an instruction is renamed, a steering algorithm decides the target cluster for that instruction. Then, if a value is consumed by a cluster other than its producer, a *copy* instruction is artificially generated by the steering logic, and inserted into the ROB and the issue queue of the producer cluster. Copy instructions are issued to a network connecting all clusters, and their execution time depends on the interconnect architecture and the distance between the source and destination clusters. The inter-cluster communication latency has a critical impact on global performance [4], 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, as shown in this paper, 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



a) Processor architecture with four clusters. b) Cluster

c) Processor Front-End

Fig. 1. Baseline microarchitecture block diagram.

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 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 might insert instruction replicas into several clusters. Experimental results show a considerable reduction of copy instructions, which leads to average performance speedups between 3% and 15% for different bus-based interconnects, and between 3% and 7% for the evaluated point-to-point networks, while still reducing the global dissipated energy by up to 12%.

The remainder of this paper is organized as follows. Section 2 describes the baseline clustered architecture. Section 3 introduces the proposed algorithm, whose hardware implementation is described in Section 4. Sections 5 and 6 show an experimental evaluation of performance and power, respectively, and Sections 7 and 8 present some related work and concluding remarks.

2 Microarchitecture Overview

This section presents the baseline clustered architecture used to implement and evaluate our proposal. To guarantee that our reported performance gains add up on existing common architectural improvements, a sophisticated baseline design has been modeled, using a trace cache, an advanced steering algorithm, and non-trivial interconnection network topologies. These features have been proposed in previous research, and are summarized in this section to aid in the understanding of the rest of the paper.

The baseline architecture is a superscalar, single-threaded, clustered processor, whose block diagram is represented in Figure 1. The processor front-end fetches x86 macroinstructions, decodes them, and dumps the generated microinstructions (μ-ops) into a μ-op queue (Figure 1a). Then, uops are dispatched into a shared ROB, which tracks their global program order until they commit. Memory uops reserve an entry in a global LSQ, while the rest of them are steered to the clusters and reserve a local IQ entry. The LSQ implements the load bypassing and load forwarding optimization techniques, and its shared design guarantees a global ordering of memory accesses. When a non-speculative uop at the ROB head completes, it is processed by the fill unit. This component builds a temporary trace that is eventually sent back to the trace cache.

After an uop is dispatched, a steering algorithm decides its target cluster. When an arithmetic uop is steered into a cluster lacking any input operand, a copy uop is generated and inserted into the IQ of some of the clusters containing the required operand. A shared register alias table (RAT) stores the register mappings for each cluster; it is indexed by a logical register identifier, and returns the private physical register associated in each cluster, or a void label if the value is not present for that cluster. Finally, each cluster contains a private IQ, functional unit pool, and physical register file (Figure 1b).

The processor front-end of the baseline architecture uses both a trace cache and an instruction cache (Figure 1c). The trace cache contains sequences of predecoded uops, and on a hit, the contents of the accessed line are directly copied into the uop queue. The trace cache is looked up in parallel with the instruction cache. On a trace cache miss, the fetched instruction cache line is decoded, and the generated uops are then inserted into the uop queue. A successful access to the trace cache has two main benefits: on one hand, the decode latency is avoided; on the other hand, a taken branch does not prevent subsequent uops from being fetched in the same cycle. Thus, a trace cache based front-end is an effective solution to increase fetch bandwidth, especially in architectures implementing a CISC instruction set, like the x86 ISA.

2.1 Trace cache

The trace cache is indexed by the program counter (*eip* register) and a sequence of bits representing the predicted behavior of the next branches. On a hit, the trace cache returns a sequence of pre-decoded uops that can be directly dispatched without further processing. This model relies on a multiple branch direction prediction, that is, a branch predictor with the capability of providing in a single cycle several predictions, each based on the original program counter and the previous prediction [5].

The trace cache is organized as a set-associative structure storing trace lines. The implementation used through the experiments is based on the original proposal by Rotenberg et al. [6], where the fields of each trace cache line are the listed in Table 1. Traces of non-speculative uops are stored after the commit stage in the fill unit, which is a temporary buffer of the same size as the maximum trace size. When a trace is full, a new trace line is allocated, replaced, or updated

TABLE 1
Trace line fields (N = trace size, B = maximum number of branches per trace, c = number of clusters, U = number of bits used to represent an instruction).

Name	Size (bits)	Meaning
`valid`	1	If set, the trace line contains a valid sequence of uops.
`tag`	32	Trace starting address.
`uop_num`	$\lceil log_2 N \rceil$	Number of uops in the trace.
`branch_mask`	B	Bit mask with as many 1s as branches in the trace, omitting the last branch if it is the last uop in the trace.
`branch_flags`	B	Bitmap representing the direction of the branches in the trace. Only those bits set to 1 in `branch_mask` are valid in `branch_flags`.
`fall_through`	32	Address of the next instruction to fetch after the trace.
`target`	32	If the last uop in the trace is a branch, address of the next instruction to fetch in case it is taken.
`uop_list`	$N \cdot U$	List of uops forming the trace.

TABLE 2
Topology-aware steering.

If *imbalance* ≥ *threshold* (= num. clusters * 8):
    ❶ Exclude clusters with *workload* > 0.
If each input operand is *available* in some cluster in the machine:
    ❷ Exclude clusters that do not minimize the longest communication distance to copy not *present* operands.
Else:
    ❸ Exclude clusters that do not maximize the number of *present* operands produced in that cluster.
❹ Exclude clusters that do not minimize the *workload* counter.

in the trace cache, and the contents of the fill unit are copied into it. It is possible to obtain final traces smaller than the maximum trace size for the following reasons: on one hand, the number of branches within a trace is limited (mainly by the multiple branch predictor width); on the other hand, uops belonging to the same x86 macroinstruction are not allowed to span several traces. Thus, the fill unit might be drained before its maximum occupancy is reached.

2.2 Steering Algorithm

A major design issue of a clustered microarchitecture is the steering algorithm. After dispatching arithmetic instructions, the steering algorithm decides which cluster they are sent to, aiming at trading off workload balance among clusters and inter-cluster communication for input dependences satisfaction. Likewise, the steering algorithm decides from which cluster to copy an uop's input operand in case it is not present. In the modeled clustered architectures, a slightly modified version of the *topology-aware* steering algorithm [7] is implemented, which works as follows.

Before steering an uop, an initial set of candidates is created including all clusters. This set is progressively reduced by applying four successive filters and discarding inadequate candidates, and finally a random cluster is selected among the resulting set. The actions performed by each filter are listed in Table 2. Filters 1 and 4 aim at balancing the workload among clusters, while filters 2 and 3 try to reduce the communication latency due to absent input operands.

Regarding the workload balance, each cluster tracks the number of instructions dispatched to it (di_i), and the average of these counters is updated globally (di_{avg}). Then, each cluster computes a private *workload* counter as the deviation of dispatched instructions $d_i - di_{avg}$, and a global *imbalance* counter is calculated as the maximum absolute value of the *workload* counters. Filter 1 in Table 2 is applied when *imbalance* exceeds a *threshold*, which has been empirically set as the number of clusters multiplied by 8. Regarding uop input dependences, an operand is said to be *present* in a cluster when there is a physical register associated to it. The operand is said to be *available* when it is present and the instruction producing its value has completed.

Notice that *topology-aware* steering is a sophisticated algorithm that effectively lowers the communication among clusters, while still keeping a fair workload balance. Since the technique proposed in this paper aims at further reducing inter-cluster communication, we have opted for a powerful baseline steering for fair comparison. Less costly and efficient steering algorithms would lead to higher potential in communication reduction and higher speedups for our proposal.

2.3 Interconnection Network

Copy uops are generated in the dispatch stage when a logical register is not mapped in a cluster where it is consumed. These uops are handled as arithmetic instructions, except that they are scheduled to the interconnection network instead of a functional unit, and their source and destination physical registers are placed in different clusters. The interconnect can be viewed as a black box whose interface is formed by an injection/ejection link connected to each cluster. Once in the instruction queue (IQ) of the source cluster, a copy uop is woken up when the source register is ready and the injection link is available. The interconnection network will forward the packet to the destination cluster, and eject the contents into its register file through a dedicated write port.

In general, the communication latency of a message depends on the network topology, routing algorithm, and switching mechanism. Since inter-cluster communication occurs on chip, links can be fairly assumed as wide as the message size, which includes a 32-bit value, a 3-bit target cluster identifier (for an 8-cluster architecture), and a 6-bit target register identifier (for a 64-entry register file). As messages are very short, packet switching is used. The communication latency can be computed as $lat_0 + lat_{cont} = (T_r + T_l)d + lat_{cont}$, where lat_0 is the zero-load latency or the time required to forward the message to the destination without contention, lat_{cont}

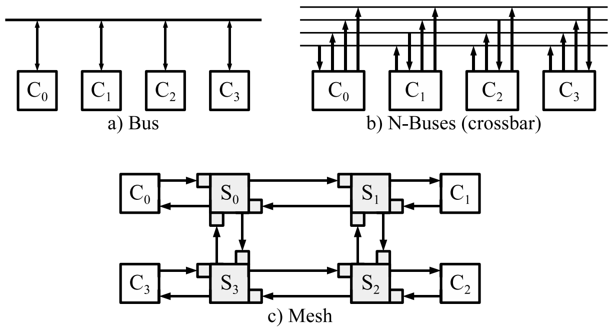


Fig. 2. Interconnection networks.

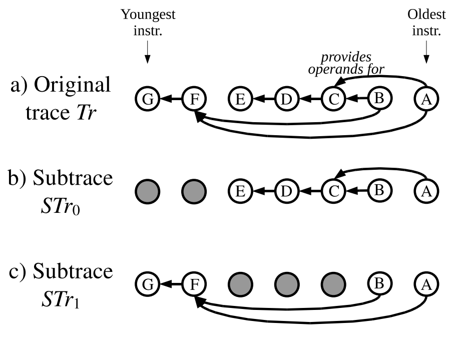


Fig. 3. Extraction of independent subtraces.

is the total contention delay due to packet collisions, T_r is the routing and forwarding time, T_l is the link transmission time (including its arbitration when it is shared), and d is the number of traversed links.

The lat_{cont} component is an unpredictable value computed through simulation, while lat_0 depends on the network topology, and router, switch and link delays. In our experiments, three network topologies have been evaluated —bus, crossbar, and 2-dimensional mesh—, with different link delays between intermediate nodes ranging from 2 to 8 cycles, intended to be representative delays for different technologies. Figure 2 represents the block diagrams of the evaluated topologies for a 4-cluster processor.

- In a bus topology (Figure 2a), no routing is used ($T_r = 0$) and all nodes are at 1-link distance from each other ($d = 1$). However, the bus transfer time T_l might grow considerably for a high number of connected clusters. We have considered lat_0 values of 2, 4, and 8 cycles.
- The crossbar topology (Figure 2b) can be viewed as c buses connecting c clusters. Each cluster can dump a message into $c - 1$ different buses, and the chosen bus depends on the message destination. The lat_0 component does not differ from the bus topology; there is no need for routing ($T_r = 0$), messages traverse only one link ($d = 1$), and T_l depends on the number of connected nodes. Again, 2, 4, and 8 cycles are considered for lat_0.
- The two-dimensional mesh (Figure 2c) requires a router attached to each cluster, which applies the XY routing algorithm after a delay of T_r cycles. The link delay (T_l) can be very low, since only point-to-point connections are used. In the largest evaluated mesh (4×2 for 8 clusters), d takes a maximum value of 4 hops. In this case, $2d$ and $4d$ cycles are considered for lat_0, which corresponds to T_r+T_l=2 and T_r+T_l=4, respectively. When connecting 3, 5, and 7 clusters with a mesh topology, 2×2, 3×2, and 4×2 meshes are assumed, respectively, where one of the end links is left disconnected in each case.

3 Generation of subtraces

Based on the architectural background presented in the previous section, the rest of this paper focuses on a novel technique for automatic extraction of parallelism at runtime. 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 3a 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. Figure 3b and 3c 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, at the expense of probably executing some instructions in both subtraces.

3.1 Proposed algorithm

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 (*STr*$_0$, *STr*$_1$, etc.) of maximum size N, where c is the number of clusters. Figure 4 represents an example using two clusters and the flow of instructions shown in Figure 3.

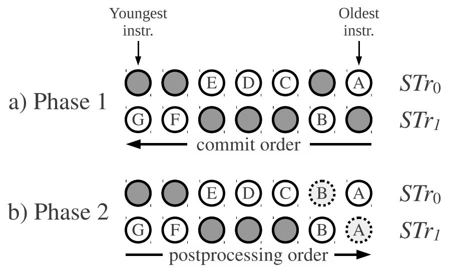


Fig. 4. Algorithm to extract subtraces.

- **Phase 1. Selection of subtrace.** As shown in Figure 4a, instructions from trace *Tr* are first split individually among subtraces *STr*$_0$ and *STr*$_1$. When an uop commits, it is assigned to that subtrace containing its input operands. If the input operands are either contained in different 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. Similarly to the steering algorithm, the subtrace imbalance counter is based on subtrace length deviations (see Section 2.2), and the optimal subtrace imbalance threshold has been found to be equal to 8 in this case.

- **Phase 2. Satisfying dependences.** 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 4b, 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 recursively 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*$_0$.

Notice that memory instructions are excluded from replications, since the LSQ is a global structure. For simplicity, *load* and *store* instructions are treated equally as arithmetic instructions in the subtrace generation process and the subtrace storage in the trace cache, but the replication information attached to them is ignored after they are dispatched. One single copy of every memory instruction is inserted into the global LSQ, and thus no extra pressure is incurred on the memory hierarchy when the number of clusters or the instruction replication increases.

4 Hardware Implementation

4.1 Instruction Numbering

To identify instruction dependences after the commit stage, an instruction numbering mechanism is proposed, which labels committed instructions with continuous identifiers as per program order, and stores for each instruction the identifiers of its input dependences. These identifiers are called *sequence numbers* and are denoted as $\mathtt{I}_{seq}$ or $\mathtt{D}_{seq}$, where `I` is an instruction and `D` is one of its input dependences.

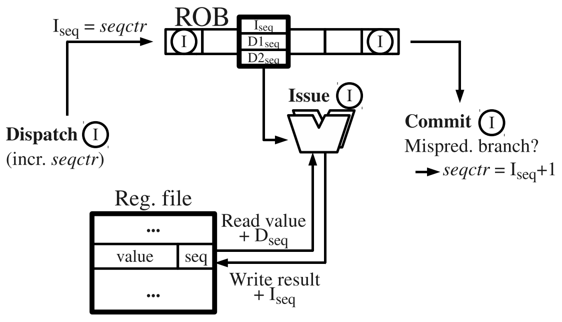


Fig. 5. Sequence number assignment.

Figure 5 represents the process of sequence number assignment. Each ROB entry contains three sequence numbers, one for the contained instructions and two for its input dependences. Likewise, each register file entry holds the sequence number of the instruction that produced the associated value. A global counter *seqctr* is increased and assigned to $\mathtt{I}_{seq}$ when instruction `I` is dispatched. When `I` is issued, $\mathtt{D1}_{seq}$ and $\mathtt{D2}_{seq}$ are read from the register file jointly with the source operand values, and $\mathtt{I}_{seq}$ is written into the destination physical register when the operation finishes.

When `I` commits, all sequence numbers are available to be used in the fill unit. If `I` is a mispredicted branch, the content of the ROB is squashed, and *seqctr* is set to $\mathtt{I}_{seq}$+1. By assigning sequence numbers with this algorithm, a continuous range of instruction sequence numbers is guaranteed, which has the convenient property of providing a direct correspondence between instructions and their position within the fill unit (thus avoiding associative ports for searches).

4.2 Fill Unit

In a trace cache processor, the fill unit is a hardware structure placed after the commit stage aimed at reconstructing a trace of committed uops and storing it in a new trace cache line. In the proposed architecture, the fill unit is additionally in charge of implementing the subtrace generation algorithm. This hardware structure consists of an N-entry buffer (N is the maximum trace length). Each buffer entry contains three fields: the associated instruction bits, a c-entry bitmap representing the presence of an instruction in each subtrace (c is the number of clusters), and the sequence numbers of the instruction and its input dependences.

The algorithm phases are implemented with two different operations in the fill unit, referred to as *fill-up* and *emptying* processes, respectively. In the former, committing instructions are assigned to one single subtrace each, while in the latter, subtraces are made independent by replicating instructions to satisfy the required input operands. Each process has an associated pointer, referred to as *fillhead* and *emptyhead*, initially pointing to buffer positions 0 and $N-1$, respectively. Below, the algorithm implementation is described using the example shown in Figure 6.

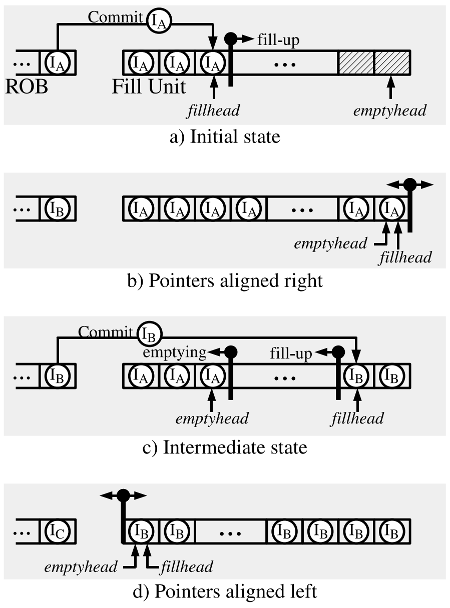


Fig. 6. Subtrace generation algorithm implemented in the fill unit.

Initially, the fill unit starts the *fill-up* process with the first N instructions extracted from the reorder buffer (ROB) at the commit stage (Figure 6a). The initial trace is referred to as trace A, and associated instructions are represented as $\texttt{I}_A$. When $\texttt{I}_A$ commits, it is inserted into the fill unit at the position pointed to by *fillhead*, and this pointer is incremented. In the associated entry, the c-entry bitmap is updated by activating the bit corresponding to the initial subtrace assignment.

Once the buffer fills up (Figure 6b), trace A is completely held in the fill unit. At this time, phase 1 of the subtrace generation algorithm is complete for trace A, and each entry's bitmap associates one instruction with one single subtrace. The *emptying* process is now activated for trace A, and the *fill-up* process starts for trace B, which is formed of the next N instructions $\texttt{I}_B$ placed in the ROB. From now on, both the *fillhead* and *emptyhead* pointers are moved from right to left, i.e., decreased.

In a subsequent intermediate state (Figure 6c), the *fill-up* process continues for trace B, in which instructions $\texttt{I}_B$ are taken from the ROB and placed into the fill unit. At the same time, the *emptying* process works on trace A, implementing phase 2 of the subtrace generation algorithm. For each emptied instruction $\texttt{I}_A$, the stored sequence numbers of its input dependences $\texttt{D}_A$ are looked up. Then, if the buffer entries pointed by these sequence numbers are valid (i.e., are within the *emptying* range), instruction $\texttt{D}_A$ is replicated in all subtraces where its producer $\texttt{I}_A$ is present. For this aim, the c-entry bitmap of $\texttt{D}_A$ is updated by activating those bits which are set in $\texttt{I}_A$'s bitmap (*or* operation). Each instruction extracted from the fill unit is sent to the trace cache, where the generated independent subtraces are stored.

When trace A finishes the *emptying* process (Figure 6d), phase 2 of the subtrace generation algorithm is complete for this trace, and the c generated independent subtraces are stored into the corresponding trace cache line. When additionally the *fill-up* process for trace B completes, the direction of the *fillhead* and *emptyhead* pointers is switched again. The fill unit starts the *emptying* process for trace B, and trace C of instructions $\texttt{I}_C$ waiting at the ROB head begin to enter the fill unit from left to right.

5 Experimental Evaluation

TABLE 3
Baseline machine parameters.

Processor Core	
Decode, dispatch, steer, commit bandwidth	8 uops/cycle
Issue width	2 uops/cycle (each cluster)
Trace cache	256 traces (64 sets, 4 ways), 16-uop traces
Global storage resources	256-entry ROB, 64-entry LSQ
Private resources per cluster	40-entry IQ, 92-entry RF
Functional units per cluster and latency (total/issue)	4 Int. add (2/1), 1 Int. mult. (5/5), 1 Int. div (20/10) 1 FP add (5/5), 1 FP mult. (10/10), 2 FP. div. (20/20)
Branch predictor type	4-way hybrid (2-level + bimodal) 2-level pred.: 8-bit history, 1-entry L1, 1K-entry L2. Bimodal pred.: 1K 2-bit counters. Choice pred.: 1K entries.
Memory Hierarchy	
L1 cache	32KB, 2-way, 64-byte block, 2-cycle latency
L2 cache	512KB, 8-way, 64-byte block, 10-cycle latency
Main memory	100-cycle access time.

This section presents a performance evaluation of the proposed techniques. Experiments have been carried out on top of the Multi2Sim 2.2 simulation framework [8], a cycle-accurate simulator for $x86$-based superscalar processors, modified to model a clustered architecture, inter-cluster network topologies, and independent subtraces generation. The simulator accurately tracks the processor pipeline state cycle by cycle, as well as the memory hierarchy including a trace cache. The parameters of the modeled machine are summarized in Table 3.

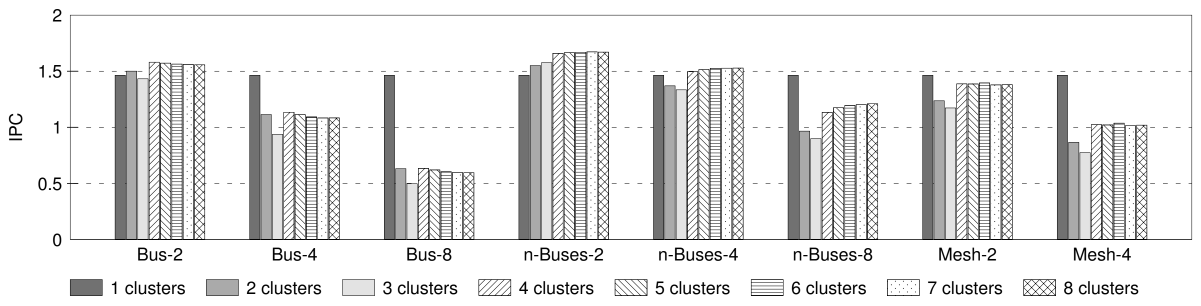
The Mediabench [9] suite has been used as a workload to evaluate the devised techniques. These applications include image and video processing, audio encoding, or speech recognition, among others. The Mediabench suite provides a particular potential for dynamic extraction of parallelism, since it includes extremely parallel algorithms whose implementation is based on a traditional sequential programming model. This is in contrast to the SPEC CPU benchmarks, which do not provide high amounts of intrinsic parallelism, or to SPLASH/Parsec benchmarks, which exploit their parallelism mostly at compile time using the *pthreads* programming model. The presented results include partial program executions, where simulations are stopped after the first 100 million uops commit.

5.1 Performance Evaluation

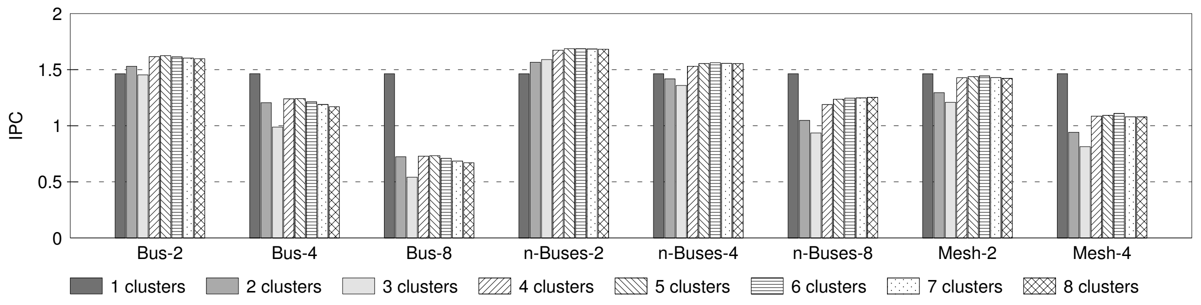
Figure 7 shows a performance study of the proposed technique, including performance results for the baseline architecture (7a), performance for a clustered processor with automatic generation of subtraces (7b), and the resulting performance speedup (7c). The number of clusters ranges from 2 to 8, and the evaluated inter-cluster networks are a *bus* (with 2, 4, and 8 cycles for lat_0, see Section 2.3), a *crossbar* (or *n-buses* with 2, 4, and 8 cycles for lat_0), and a mesh (with 2 and 4 cycles lat_0)[1]. Each bar represents the average speedups for the whole benchmark suite (16 workloads).

- In a bus topology, the fastest configuration (lat_0=2) provides speedups below 5%. However, a fast bus is only suitable for a very low number of connected clusters. When the zero-load latency value 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.

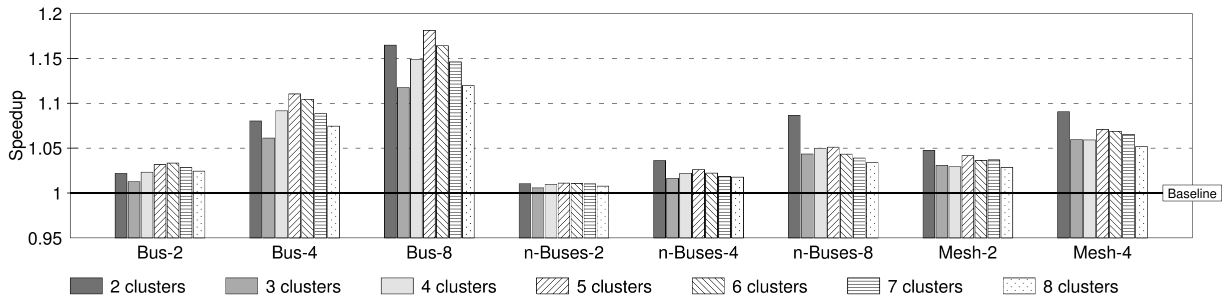
1. Notice that all combinations for these values are represented for performance trends observability, even if some of them might be impractical for a real system. For example, a 2-cluster mesh, simulated as an indirect network with 2 intermediate switches, could be easily replaced by a 2-cluster crossbar (i.e., 2 unidirectional links), reducing the communication latency and the implementation cost.



a) Performance on baseline clustered processor.



b) Performance for clustered processor using automatic subtraces generation.



c) Performance speedups when generating subtraces.

Fig. 7. Performance for different network topologies and 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 a 4-cycle zero-load latency provides speedups above 5%, regardless of the number of clusters.

To provide some insight into each specific benchmark's behavior, performance has been evaluated individually for each benchmark for some specific configurations, as plotted in Figure 8. Shorthand *c2-bus2-base* stands for 2 clusters/2-cycle bus/baseline machine; *c8-bus8-subtr* means 8 clusters/8-cycle bus/machine with subtraces generation; etc. The bus latency values chosen for this plot intend to be representative for the corresponding number of clusters in each configuration, also resembling those used in previous works on clustered microarchitectures [7].

Performance values are given as a global IPC, i.e., number of instructions per cycle committed from the global ROB. Notice that an increase in the number of clusters might result in an IPC loss in some cases (Figures 7 and 8), which under identical circumstances would negate the reason for using more than one cluster. However, clustering the major microprocessor components (e.g, register files or IQs) helps for a reduction of the clock cycle time, which should result in lower global execution time. The analysis of clustered microarchitectures and their benefits have been widely studied in the literature and are now out of the scope of this work. In this work. We assume these benefits and focus on the additional performance gains obtained from subtraces.

5.2 Copy instructions and replicated uops

When independent subtraces are dispatched, the number of copy uops decreases, reducing communication and network contention. Instead, uop replicas are dispatched, which are usually faster (2 cycles for very frequent integer additions or

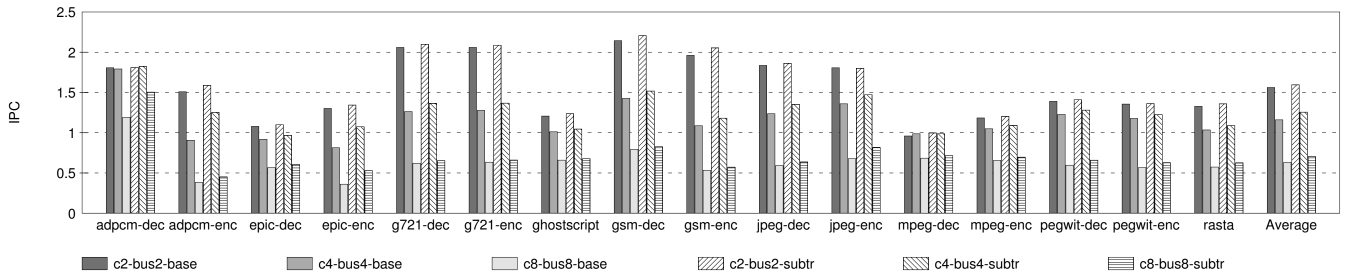


Fig. 8. IPC values for individual benchmarks and specific designs.

effective address computations), and cause dependent instructions in the IQ to be issued earlier. Figure 9 shows the number of copy instructions (*copy_base* for the baseline machine and *copy_subtr* for the machine implementing independent subtraces) and the number of replicated uops, represented as an average fraction over the total number of committed uops.

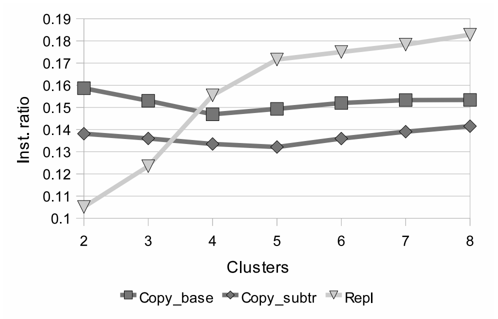


Fig. 9. Copy and replicated uops.

These values have been obtained for a *bus2* network, though the interconnect topology does not considerably affect the aspect of the plotted curves. When an uop at the ROB head with several replicas commits, one of them is tracked as the original uop, and the rest of them are counted as replicas. Results show that the number of copy uops decreases by about 10% in any configuration. On the other hand, uop replicas range from 10% to 20% of the total committed uops, depending on the number of clusters.

The results presented in this section partially explain the performance speedups shown in Figure 7. Since inter-cluster communication acts most of the time as a performance bottleneck, execution time is reduced according to the decrease in copy instructions. Thus, the points that show a shorter distance between the *copy_base* and *copy_subtr* curves correspond to the highest performance speedups.

5.3 Impact of the Trace Size

We have measured the impact of different trace sizes, ranging from 8 to 64 uops. Figure 10 shows the speedup achieved by the proposed technique executed on top of a 4-cluster processor model with the *bus4* interconnect topology. For the sake of clarity, only three benchmarks are plotted, including the average curve for the whole Mediabench suite. The optimal trace size is observed at different positions for each benchmark, mostly depending on the parallelism exhibited by each.

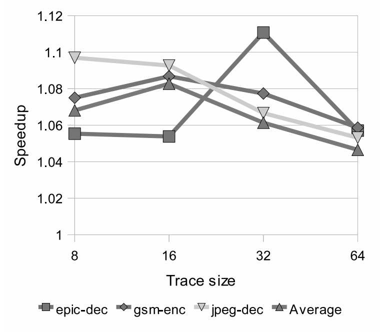


Fig. 10. Speedups for different trace sizes.

The trace size has two main implications on performance. On one hand, a larger trace increases the chance of extracting existing parallelism within a given trace with a lower uop replication rate. On the other hand, a trace cache with the same number of larger cache lines makes the hit ratio shrink dramatically. In our benchmark suite, the average fraction of committed uops fetched from the trace cache is 57.9%, 52.2%, 41.7%, and 27.3% for 8-, 16-, 32-, and 64-entry traces, respectively. The *Average* plotted curve represents the whole Mediabench suite, and shows an optimal average trace size of 16 entries (used for our baseline machine).

5.4 Impact of the Sequence Number Length

During the *emptying* process implemented in the fill unit, each instruction `I` is extracted and dumped into the trace cache. For each dependent instruction `D1` and `D2` present in the fill unit, bitmaps are updated to determine their future presence in subtraces. The index of `D1` and `D2` are determined with sequence numbers attached to `I`, whose length should be determined as a trade-off between performance and hardware cost. If the sequence numbers are too short, they might generate aliasing by detecting false dependences. On the contrary, too large sequence numbers increase the fill unit entry size. In any case, sequence number aliasing causes false dependence detections, but never leaves a real dependence undetected. Since the sequence number is used to index the N-entry fill unit, the sequence number size should not be lower than $log_2 N$ bits.

When false dependencies are introduced, independent subtraces become larger than necessary, and redundant computations are performed across clusters. In Figure 11, the average subtrace size is represented for the studied benchmarks,

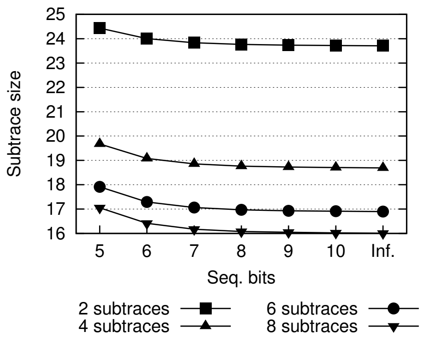


Fig. 11. Average subtrace length for different sequence number lengths.

TABLE 4
Power consumption and total energy dissipated on average for MediaBench on a 4-cluster processor with a *bus-4* inter-cluster network.

		Baseline			Subtraces		
		Lk. pwr (mW)	**Dyn.Pwr (mW)**	**Total E. (mJ)**	**Lk. pwr (mW)**	**Dyn.Pwr (mW)**	**Total E. (mJ)**
Front-end		376.3	909.7	19.6	376.3	953.6	18.2
Mem. subsystem		1286	229.6	24.9	1286.0	239.4	23.0
Traces Support		40.1	4.1	0.7	44.5	11.5	0.8
Renaming		3.1	311.6	4.6	3.1	326.9	4.3
Network		0.9	19.4	0.3	0.9	18.2	0.3
Per-cluster structs	**RFs**	13.9	109.2	1.9	13.9	129.9	1.9
	IQs	12.7	126.5	2.1	12.7	149.4	2.2
	FUs	5666.1	2599.0	137.0	5666.1	2758.4	127.7
Total		7399.2	4308.9	191.2	7403.5	4587.42	178.4

varying the number of clusters between 2 and 8, and using original traces of $N = 32$ μ-ops. On one hand, it is observed that a higher number of clusters (subtraces) provides a smaller average subtrace length. The reason is that traces have the means to be further split into several independent subtraces and still reduce their length by exploiting their inner ILP. On the other hand, results show that a sequence number length of 8 bits almost completely palliates false dependences detection, since the average subtrace length is very similar to an ideal design with an unbounded sequence number length.

6 Power and Energy Study

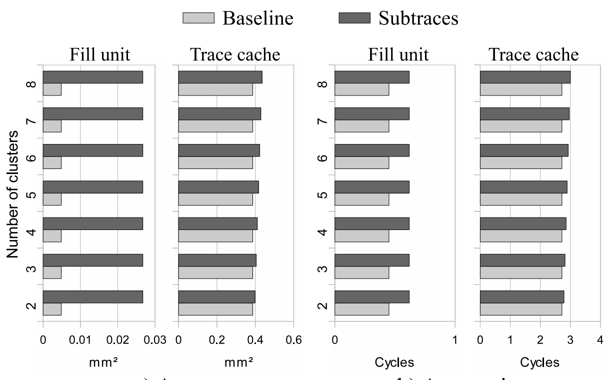
A detailed power and energy study has been performed to compare the technique proposed in this paper with the baseline machine. The results shown in this section have been obtained with the McPAT 0.7 tool [10], a power, area, and timing model of a complete processor pipeline and memory hierarchy, which is based on the Cacti code [11] to compute statistics related to data arrays and CAM structures. McPAT provides detailed power consumption statistics for each hardware component, including sub-threshold leakage, gate leakage, and run-time dynamic power, based on the hardware complexity of the computed designs and their access rates. The tool has been set up to model our baseline and proposed schemes for a 45nm technology and a working frequency of 3.4GHz, and it has been extended with the following features:

- A multi-cluster architecture model has been added to McPAT by replicating those structures private per cluster (register files, instruction queues, functional units, and results broadcast buses). The hardware characteristics and access statistics of these components are provided by Multi2Sim and summed up to obtain the total power consumption. By using the configurable NoC implementation provided by McPAT, the inter-cluster network has been modeled and included in the global power results.
- Relying on the Cacti code, two trace cache models have been added to McPAT: a baseline trace cache model with the cache line fields presented in Table 1, and the proposed trace cache model where an additional bitmap is attached to every uop indicating the presence in subtraces. Both trace cache models have a capacity of 256 traces, a 4-way associativity, and a trace size of 16 uops. We have made sure that the incurred hardware overhead does not impact the global processor cycle time.
- Likewise, Cacti has been used to obtain two fill unit models, accepting as many instructions per cycle from the ROB as the commit width w. The baseline fill unit has been modeled as a 16-entry buffer with w write and w read ports. This complexity increases for the proposed fill unit: for the *fill-up* process, w write ports are required in the buffer, indexed by the *fillhead* pointer; for the *emptying* process, w read ports are used to extract w instructions at the location pointed by *emptyhead*. Moreover, $2w$ additional write ports are required to update the presence of at most 2 input dependences per instruction. In total, the proposed fill unit is implemented with w read + $3w$ write ports. Each fill unit entry is $c + 56$ bits large (32 uop bits + three 8-bit sequence numbers + c-entry bitmap), being c the number of clusters.

6.1 Detailed Power Consumption

Table 4 shows detailed power consumption values for a specific processor configuration with 4 clusters and a *bus-4* network topology. For both the baseline and the proposed machines, the columns include leakage power, dynamic power, and the total energy dissipated on average for the execution of the Mediabench suite. Each row represents a set of hardware structures, including the processor front-end (BTB, branch predictor, decoder, uop queue), the memory subsystem (TLB, data cache, instruction cache, L2 cache, LSQ), traces support (trace cache, fill unit), register renaming structures (front-end RAT, retirement RAT, free list), inter-cluster network, register files, instruction queues, and functional units (including integer and floating-point ALUs, and result broadcast buses).

As observed, the column representing the leakage power does not vary for the baseline and proposed machines, except for the hardware devoted to subtraces generation. The joint leakage power consumption of the trace cache and fill unit (traces support) increases by 10.1% in the proposed scheme. The dynamic power increases for all processor components due to a higher transistor switching activity, except for the inter-cluster network, whose activity is lowered by a reduction of the number of copy instructions. In total, the dynamic power consumption increases by 6.5%. However, this increase is compensated by a reduction of the execution time, which



a) Area b) Access time

Fig. 13. Area and timing results for the trace cache and fill unit.

leads to a global energy saving of 6.7% (bottom-right cell in the table).

6.2 Energy Savings

The behavior exhibited by the specific architecture presented in Table 4 is representative for all evaluated designs. Figure 12 shows the energy savings provided by each design with a different inter-cluster network and number of clusters, where each bar represents the average savings for the Mediabench suite.

There is a tight relationship between the speedups shown in Figure 7 and the energy savings. Though some components suffer a higher leakage consumption and dynamic energy per access in a processor with automatic subtraces generation, the execution time is reduced in such a way that the total spent energy in the processor core decreases (up to 12.2% in some configurations). There are some cases with small speedups that show an energy saving close to 0% (*n-Buses-2* network), but there is no simulation showing a higher energy dissipation for the proposed scheme compared to its homologous baseline processor, showing the proposed architecture as an energy-efficient approach.

6.3 Area and Timing

The Cacti tool [11] has been used to measure the area (Figure 13a) and timing (Figure 13b) overheads incurred by the subtraces support in the fill unit and trace cache for processors with a different number of clusters. While the baseline fill unit occupies about 0.005mm^2 using a 45nm technology, the proposed fill unit multiplies this area by a factor of 5.5. This considerable increase is due to the additional write ports, but it remains a negligible fraction of the total processor area (less than 0.01%). The fill unit access time increases by about 40%, but it is still low enough to complete in one single cycle for the 3.4GHz modeled processor. Additionally, there is a slight increase in the proposed fill unit area and access time for a growing number of clusters due to the bit mask associated to each fill unit entry. This overhead is negligible and cannot be visually appreciated in the figures.

Regarding the trace cache, the area and access time increase incurred by the proposed design is induced by the bit mask attached per uop in every stored trace. In this case, the number of clusters has a noticeable impact on the proposed trace cache area and access time, because there is a considerable number of uops affected by the associated bit mask size. The area overhead ranges from 3.1% (2 clusters) to 12.6% (8 clusters), while the access time increase lies between 2.5% (2 clusters) and 10.4% (8 clusters). However, the trace cache access time does not exceed 3 cycles even for the worst case.

7 Related Work

The fact that performance in a clustered architecture is very sensitive to the inter-cluster communication latency has been widely discussed in previous research works [4][12]. Likewise, a large amount of research has focused on steering algorithms trying to balance clusters' workload, while at the same time minimizing the number of copy instructions [13][14][12][7].

In [13], Baniasadi and Moshovos propose several steering heuristics, classified as static and adaptive. They propose a relatively simple method that offers a competitive performance by just changing the target cluster every three instructions. The work by Canal et al. [14] focuses on dynamic run-time heuristics on heterogeneous two-cluster processors, where only one of the clusters is able to execute floating-point instructions. In [12], Parcerisa and González show how to reduce both communication and workload imbalance by applying value prediction. Finally, the *topology-aware* steering, presented in [7] and adopted for the present work, considers complex networks and a higher number of clusters. However, all these proposals are constrained by the ILP present in the executed sequential code, which imposes a lower bound in the number of generated copy instructions for a given workload balance.

On the other hand, the trace cache was proposed as an effective solution to fulfill high fetch bandwidth requirements in superscalar [6] and clustered processors [15][16], and was implemented in the Intel Celeron, Pentium 4, Pentium D, and Xeon microprocessor families [17]. The trace cache fill unit is a latency-tolerant component [18][19], that provides a good chance for new optimizations. In [15], a trace preprocessing is proposed in the fill unit, used later by the steering logic to dispatch dependent instructions of the same trace to the same cluster. In [16], this analysis is improved by spotting inter-trace dependences in order to place dependent instructions from different traces in the same cluster. Unlike this work, none of these optimizations consider a replication of instructions to further overcome communication delays.

The idea of introducing artificial parallelism through replication has been also employed by Madriles et al. [20]. In this case, additional TLP (thread-level parallelism) is induced by the compiler, which replicates basic blocks to create speculative threads. These threads are created by first representing data- and control-dependent basic blocks with a directed graph, and then applying the *multilevel graph partitioning* algorithm on it [21]. An architectural design is also proposed to undo the execution of mispredicted threads.

Compared to previous works, our proposal provides three main advantages: i) it provides binary compatibility with existing sequential code, since it neither involves the compiler

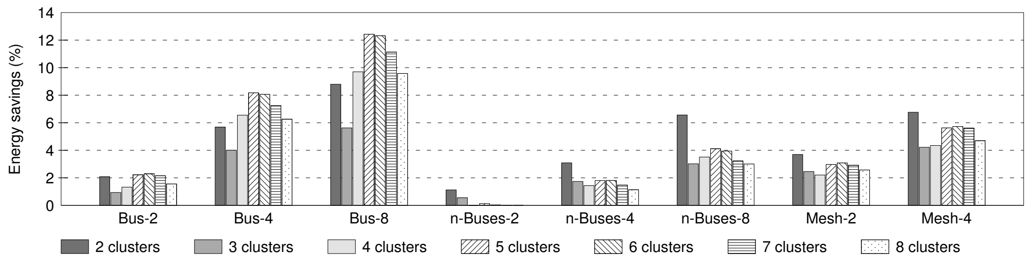


Fig. 12. Energy savings for different network topologies and number of clusters.

nor modifies the ISA, ii) the subtraces generation algorithm is lightweight enough to be implemented in hardware without incurring extra latency on the critical path, and iii) there is no need for an additional recovery mechanism, since mispredicted subtraces are not handled exceptionally. In summary, to the best of our knowledge, our proposal is the first fully hardware-based, binary-compatible approach that generates parallel subtraces out of sequential code at the instruction level.

8 Conclusions

This paper has presented an energy-efficient hardware mechanism that automatically detects independent subtraces of instructions in a sequential program. An implementation of the mechanism on top of a clustered microarchitecture has been devised, using a trace cache with a modified fill unit. By carefully replicating the execution of specific instructions, inter-cluster communication is reduced, network traffic and collisions are decreased, and global performance is benefited. When designing a clustered architecture that boosts single-thread performance at lower hardware costs, additional benefits are shown to be reached with subtraces generation for different number of clusters and interconnect topologies.

Experimental results show average performance speedups of about 3%, 7%, and 15% (accompanied by 2%, 6%, and 10% average energy savings) for the bus-based interconnects with 2, 4, and 8 transmission cycles, respectively. The evaluated point-to-point networks provide average performance speedups of about 3% and 7% (with 2% and 5% average energy savings) for 2- and 4-cycle link delays, respectively. While higher levels of performance can be achieved running multithreaded applications on multicore processors or data-parallel programs on GPUs, they necessarily go through a recoding process following a less intuitive parallel programming models. Our proposal takes especial advantage of those highly parallel applications implemented under a sequential programming model, whose intrinsic parallelism was not completely exploited at compile time.

As future work, we plan to extend the proposed technique to a multicore environment, where a reduction of inter-core communication may benefit the memory hierarchy and coherence actions. In this environment, an alternative hardware implementation should be proposed, which does not rely on a shared processor front-end with a common trace cache.

References

[1] C. McNairy and R. Bhatia. Montecito: A Dual-Core, Dual-Thread Itanium Processor. *IEEE Micro*, 25(2), 2005.

[2] R. Kalla, B. Sinharoy, and J. M. Tendler. IBM Power5 Chip: a Dual-Core Multithreaded Processor. *IEEE Micro*, 24(2), 2004.

[3] S. Palacharla, N. P. Jouppi, and J. E. Smith. Complexity-Effective Superscalar Processor. In *Proc. of the 24th Int'l Symposium on Computer Architecture*, June 1997.

[4] R. Canal, J.M. Parcerisal, and A. González. A Cost-Effective Clustered Architecture. In *Proc. of the 1999 Int'l Conference on Parallel Architectures and Compilation Techniques*, Oct. 1999.

[5] Tse-yu Yeh, Deborah T. Marr, and Yale N. Patt. Increasing the Instruction Fetch Rate via Multiple Branch Prediction and a Branch Address Cache. In *Proc. of the 7th ACM Conference on Supercomputing*, 1993.

[6] E. Rotenberg, S. Bennett, and J. E. Smith. Trace Cache: a Low Latency Approach to High Bandwidth Instruction Fetching. In *Proc. of the 29th Int'l Symposium on Microarchitecture*, Dec. 1996.

[7] J. M. Parcerisa, J. Sahuquillo, A. González, and J. Duato. Efficient Interconnects for Clustered Microarchitectures. In *Proc. of the 11th Int'l Conference on Parallel Architectures and Compilation Techniques*, Sept. 2002.

[8] R. Ubal, J. Sahuquillo, S. Petit, and P. López. Multi2Sim: A Simulation Framework to Evaluate Multicore-Multithreaded Processors. In *Proc. of the 19th Int'l Symposium on Computer Architecture and High Performance Computing, www.multi2sim.org*, Oct. 2007.

[9] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems. In *Proc. of the 30th Int'l Symposium on Microarchitecture*, Dec. 1997.

[10] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi. McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures. In *Proc. of the 42nd Int'l Symposium on Microarchitecture*, Dec. 2009.

[11] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi. CACTI 6.0: A Tool to Model Large Caches. Technical report, School of Computing, University of Utah, 2007.

[12] J. M. Parcerisa and A. González. Reducing Wire Delay Penalty through Value Prediction. In *Proc. of the 33rd Int'l Symposium on Microarchitecture*, Dec. 2000.

[13] A. Baniasadi and A. Moshovos. Instruction Distribution Heuristics for Quad-Cluster, Dynamically-Scheduled, Superscalar Processors. In *Proc. of the 33rd Int'l Symposium on Microarchitecture*, Dec. 2000.

[14] R. Canal, J.M. Parcerisal, and A. González. Dynamic Cluster Assignment Mechanisms. In *Proc. of the 6th Int'l Symposium on High Performance Computer Architecture*, Jan. 2000.

[15] D. H. Friendly, S. J. Patel, and Y. N. Patt. Putting the Fill Unit to Work: Dynamic Optimizations for Trace Cache Microprocessors. In *Proc. of the 31st Int'l Symposium on Microarchitecture*, Nov. 1998.

[16] R. Bhargava and L. K. John. Improving Dynamic Cluster Assignment for Clustered Trace Cache Processors. In *Proc. of the 30th Int'l Symposium on Computer Architecture*, June 2003.

[17] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyler, and P. Roussel. The Microarchitecture of the Pentium 4 Processor. *Intel Technology Journal*, Feb. 2001.

[18] D. H. Friendly, S. J. Patel, and Y. N. Patt. Alternative Fetch and Issue Policies for the Trace Cache Fetch Mechanism. In *Proc. of the 30th Int'l Symposium on Microarchitecture*, Dec. 1997.

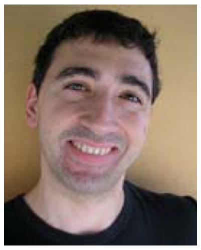
[19] Q. Jacobson and J.E. Smith. Instruction Pre-Processing in Trace Processors. In *Proc. of the 5th Int'l Symposium on High Performance Computer Architecture*, Jan. 1999.

[20] C. Madriles, P. López, J. M. Codina, E. Gibert, F. Latorre, A. Martínez, R. Martínez, and A. González. Boosting Single-Thread Performance in Multicore Systems through Fine-Grain Multithreading. In *Proc. of the 36th Int'l Symposium on Computer Architecture*, June 2009.

[21] G. Karypis and V. Kumar. Analysis of Multilevel Graph Partitioning. In *Proc. of the 7th Supercomputing*, 1995.

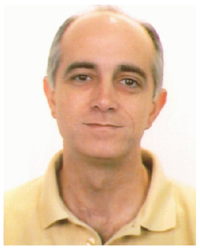
Rafael Ubal obtained his PhD Degree in Computer Engineering in 2010 from Universidad Politécnica de Valencia (Spain). He works currently as a Lecturer in the Electrical and Computer Engineering Department at Northeastern University (Boston, MA), and Postdoctoral Associate Researcher in the NUCAR group conducted by Prof. David Kaeli. His research topics of interest include power-aware cache designs, automatic parallelization of code, clustered/multithreaded/multicore architectures and GPGPU. He is the main developer of the Multi2Sim simulation framework, a CPU-GPU simulation platform for heterogeneous computing.

Julio Sahuquillo received his BS, MS, and PhD degrees in Computer Engineering from Universidad Politécnica de Valencia (Spain). Since 2002 he is an Associate Professor at the Department of Computer Engineering. He has taught several courses on computer organization and architecture. He has published over 70 refereed conference and journal papers. His research topics include multiprocessor systems, cache design, instruction-level parallelism, and power dissipation. An important part of his research has also concentrated on the web performance field, including proxy caching, web prefetching, and web workload characterization. He is a member of the IEEE Computer Society.



Salvador Petit received his PhD degree in Computer Engineering from Universidad Politécnica de Valencia (Spain). He is currently an Associate Professor in the Computer Engineering Department at the same university. His research topics include multithreaded and multicore processors, as well as memory hierarchy designs. He is a member of the IEEE Computer Society.

Pedro López is a Full Professor at the Department of Computer Engineering at Universidad Politécnica de Valencia (Spain). He received his BS degree in Electrical Engineering and his MS and PhD degrees in Computer Engineering from the same university in 1984, 1990, and 1995, respectively. He has taught several courses on computer organization and architecture. His research interests include high performance interconnection networks for multiprocessor systems and clusters and networks on chip. He has published more than 100 refereed conference and journal papers. He is a member of the editorial board of Parallel Computing journal. He is a member of the IEEE Computer Society.



José Duato received the M.S. and Ph.D. degrees in electrical engineering from Universidad Politécnica de Valencia (Spain) in 1981 and 1985, respectively. He was an Adjunct Professor with the Department of Computer and Information Science, The Ohio State University, Columbus. He is currently a Professor with the Department of Computer Engineering (DISCA) in Universidad Politécnica de Valencia. His research interests include interconnection networks and multiprocessor architectures. He has published more than 380 refereed papers. He proposed a powerful theory of deadlock-free adaptive routing for wormhole networks. Versions of this theory have been used in the design of the routing algorithms for the MIT Reliable Router, the Cray T3E supercomputer, the internal router of the Alpha 21364 microprocessor, and the IBM BlueGene/L supercomputer. Dr. Duato was a member of the editorial boards of the *IEEE Transactions on Parallel and Distributed Systems*, *IEEE Transactions on Computers*, and *IEEE Computer Architecture Letters*.