Variable-Based Multi-Module Data Caches for Clustered VLIW Processors

Enric Gibert[♦♥], Jaume Abella[♦♥], Jesús Sánchez[♥], Xavier Vera[♥], Antonio González[♦♥]

[♦]Departament d'Arquitectura de Computadors
Universitat Politècnica de Catalunya, Barcelona

[♥]Intel Barcelona Research Center
Intel Labs, Universitat Politècnica de Catalunya

email: {enricx.gibert.codina, jaumex.abella, f.jesus.sanchez, javierx.vera, antonio.gonzalez}@intel.com

Abstract

Memory structures consume an important fraction of the total processor energy. One solution to reduce the energy consumed by cache memories consists of reducing their supply voltage and/or increase their threshold voltage at an expense in access time. We propose to divide the L1 data cache into two cache modules for a clustered VLIW processor consisting of two clusters. Such division is done on a variable basis so that the address of a datum determines its location. Each cache module is assigned to a cluster and can be set up as a fast power-hungry module or as a slow power-aware module. We also present compiler techniques in order to distribute variables between the two cache modules and generate code accordingly. We have explored several cache configurations using the Mediabench suite and we have observed that the best distributed cache organization outperforms traditional cache organizations by 19%-31% in energy·delay^2 and by 11%-29% in energy·delay. In addition, we also explore a reconfigurable distributed cache, where the cache can be reconfigured on a context switch. This reconfigurable scheme further outperforms the best previous distributed organization by 3%-4%.

1. Introduction

The cache hierarchy consumes an important fraction of the total processor energy. This is even more noticeable in in-order or VLIW processors often used in the embedded/DSP domain, due to the lower complexity of the processor core as compared to out-of-order processors. For example, in the ARM10 processor family, 24% of the dynamic power is due to the data cache and 22% to the instruction cache [15]. One solution to reduce the energy consumed by cache memories is to lower their supply voltage V_{DD} and/or increase their threshold voltage V_{TH} at an expense in access time. Thus, there is a trade-off between energy consumption and performance.

Furthermore, it has been shown that heterogeneity can be exploited in several parts of the processor. A processor may contain some functional units tuned for performance and some other tuned for energy, pursuing energy efficiency [23][29]. Heterogeneity has also been explored in the memory hierarchy. For instance, Abella and González [1] proposed to divide the data cache into a fast power-hungry module and a slow power-aware module for out-of-order processors.

In this paper we also divide the data cache into two modules. This division is done on a variable basis so that the address of a variable determines the cache module where it resides. In order to do so, the address space of a process is divided into two address spaces and each one is bound to a different cache module. This scheme is applied to a VLIW processor consisting of two clusters, in which each cluster has a subset of the functional units, a local register file and a cache module. Each cache module can be set up as either a fast power-hungry module or a slow power-aware module, leading to several cache configurations with different performance and energy characteristics.

The compiler is responsible for distributing variables between the two address spaces. Once data have been distributed, memory instructions have a preferred cluster based on the accessed variables. Such preferred cluster is described by an affinity attribute. Affinities are then propagated to the rest of the instructions to guide the assignment of instructions to clusters. We develop compiler techniques to efficiently map variables to cache modules and schedule code accordingly.

The different cache configurations are evaluated for a clustered VLIW processor using *energy·delay* (ED) and *energy·delay^2* (EDD), and are compared to a clustered VLIW processor with a unified data cache. Results for the Mediabench suite demonstrate that the proposed architecture / compiler schemes outperform classical cache organizations when performance and energy are both taken into account. Furthermore, we have observed that there is not a single cache configuration that is the best for all benchmarks. Thus, we also explore a scheme in which the cache may be reconfigured on a context switch depending on the process being scheduled out and the process being scheduled in. This dynamic organization even exploits energy efficiency better.

The rest of the paper is organized as follows. In Section 2, the proposed variable-based scheme is presented, along with the different cache configurations. After that, compiler techniques are introduced in Section 3 and all the cache schemes are evaluated in Section 4. Finally, related work is exposed in Section 5, while conclusions are drawn in Section 6.

Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques (PACT'05)
1089-795X/05 $20.00 © 2005 IEEE

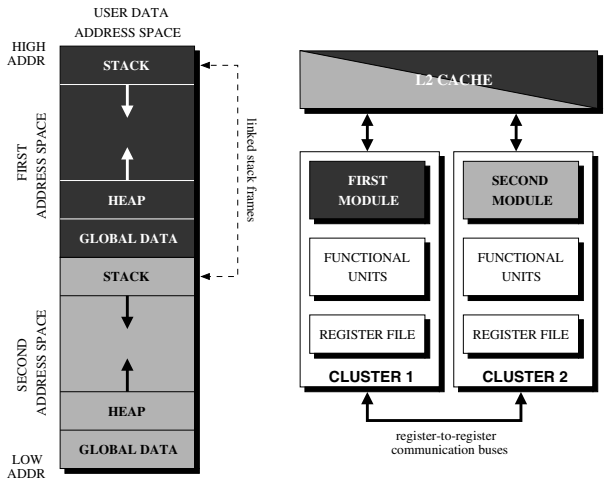

Figure 1. A variable-based multi-module cache for a VLIW processor with 2 clusters.

2. Architecture Configurations

Clustering is a technique in which the processor is divided into semi-independent units in order to overcome energy-related problems and wire delays. Each of these units is referred to as a cluster, which often consists of a local register file and a subset of the functional units. Communications inside a cluster or local communications are fast, while inter-cluster communications or global communications are slow. Hence, instructions should be assigned to clusters so that global communications are minimized while workload balance among clusters is maximized.

In this section, the proposed cache configurations for a clustered VLIW processor are presented. First, the cluster architecture and an overview of the cache organization is introduced in Section 2.1. Next, the different cache configurations are presented in Section 2.2.

2.1. A Variable-Based Multi-Module Cache for Clustered VLIW Architectures

In this paper, we propose an energy-efficient cache organization for a VLIW processor consisting of two clusters. Inter-cluster communications are achieved through a set of register-to-register communication buses or register buses for short, which are controlled by the compiler. Hence, the compiler is responsible to add and schedule an explicit copy operation whenever it assigns two register-flow dependent instructions to different clusters.

The L1 data cache is divided into two modules and each cache module is attached to a cluster. Program variables are statically distributed between the cache modules. In order to do so, the address space of a process is split into two address spaces: data or variables mapped into the first address space are always cached in the first module at runtime, while data or variables mapped into the second address space are stored in the second module at runtime. Thus, the most significant bit of a given address indicates where the datum resides. This scheme is known as a variable-based multi-module data cache and it is shown in Figure 1.

If the stack is distributed between the two address spaces, two stack pointers must be used. In this paper we do not only split the stack, but we also split individual stack frames. Hence, local variables of a given function may reside in different address spaces.

We assume a stall-on-use processor in which the processor is not stalled in case of a cache miss until the requested datum is needed. In particular, upon a miss, the processor continues executing instructions until the first consumer of the memory instruction that missed in the cache is executed. At that point, if the datum is not ready yet, the processor is stalled.

Memory instructions are statically scheduled in one of the two clusters. A memory instruction is said to be a local access when it references a datum mapped in the cache module of the same cluster. On the other hand, a memory instruction accessing data mapped in the cache module of the other cluster is referred to as a remote access.

Finally, memory disambiguation is performed locally in each cluster as long as statically-scheduled memory instructions become local accesses since each cluster stores distinct data. In the presence of a remote access, memory coherence is guaranteed by stalling the processor until the memory access is completed remotely and a reply is sent back from the remote to the local cluster. Due to the fact that remote accesses are infrequent and that there is not any communication mean between clusters other than the register buses, register buses are used to perform this request-reply transaction. Thus, when the processor detects a remote access it stalls execution. Since there may be valid values in the register buses at that time, the processor waits until the values have reached all clusters (the latency of the buses), buffers the values in temporary registers in order to continue execution after the remote access, and performs the request-reply transaction. After that, it resumes execution.

2.2. Cache Configurations

Since the data cache consumes an important fraction of the processor energy, we use heterogeneous cache modules for the proposed architecture. In particular two module types are considered: a fast power-hungry type tuned for performance and a slow power-aware type tuned for energy consumption. From these cache module types, we explore five different architectural configurations as shown in Figure 2. These configurations consist of:

- one cluster with a fast cache module and the other without any cache module, referred to as the FAST+NONE scheme shown in Figure 2(A)
- both clusters with a fast cache module, referred to as the FAST+FAST scheme shown in Figure 2(B)
- one cluster with a fast cache module and the other with a slow cache module, referred to as the FAST+SLOW scheme shown in Figure 2(C)
- both clusters with a slow cache module, referred to as the SLOW+SLOW scheme shown in Figure 2(D)

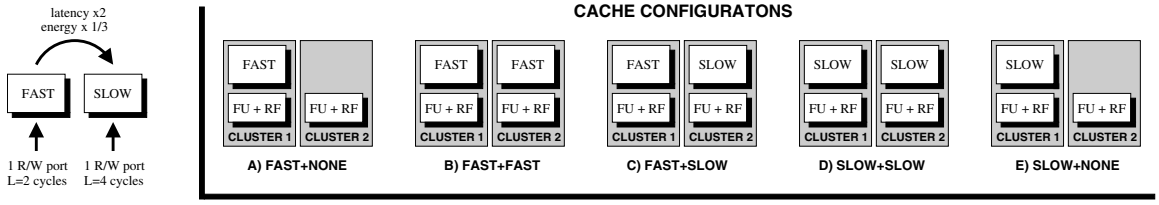

Figure 2. Cache configurations and relations between a fast and a slow modules.

- and one cluster with a slow cache module and the other without any cache module, referred to as the SLOW+NONE scheme shown in Figure 2(E)

We have assumed that each cache module has 1 read/write port. Another important consideration is the latency and power dissipation of a slow module with respect to a fast one. Processor energy can be classified as either dynamic (energy due to activity, consumed when transistors switch), and leakage (due to subthreshold leakage currents). Nowadays, leakage energy accounts for around 25% of the total energy, but trends indicate that this ratio will be soon 50% [22][26]. It is well known that decreasing the supply voltage (V_{DD}) reduces both dynamic power and leakage, and slightly increasing the threshold voltage (V_{TH}) drastically reduces leakage. However, both adjustments increase the delay. Thus, there is a trade-off between dynamic power, leakage and access time. Similarly to previous work [1], we have assumed that the latency of the slow cache should be at most 2 times larger than the latency of the fast cache. It has also been assumed that V_{DD} and V_{TH} must reduce both power sources (dynamic power and leakage) by the same percentage since optimal generic V_{DD} and V_{TH} values cannot be computed as explained in [1]. With these constraints, we have found that increasing the latency from 2 to 4 cycles reduces both power sources to around 1/3 of the fast module [24]. The characteristics of each cache module and the relations between a fast and a slow module are also depicted in Figure 2. We have used CACTI [24] to compute energy consumption.

3. Compiler Techniques

In this section, we present compiler techniques to generate code for a clustered VLIW processor with a variable-based multi-module cache. The process of mapping variables to address spaces and schedule code accordingly can be divided in several steps which are covered in deeper detail in the following sections. An overview of the process is shown in Figure 3. First, the compiler builds the Instructions-to-Variables Graph (IVG), which is a structure that represents the memory access pattern of the program, and extends it with additional information. The compiler then decides a variable mapping. Once a mapping has been computed, the affinity of memory instructions is computed and they are assigned a latency. Next, slacks are computed and affinities are propagated to the other instructions. Finally, code is scheduled and this information is fed back to the mapping algorithm in order to refine it. This iterative process finishes when the compiler estimates that no more benefit can be obtained in the trade-off between performance and energy consumption.

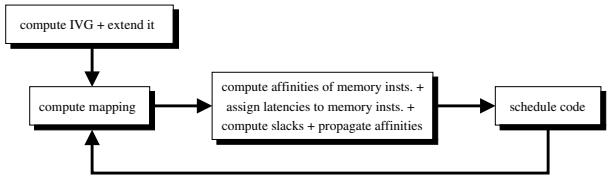

Figure 3. Overview of the compiler techniques used to map variables to address spaces and schedule code accordingly.

All this iterative procedure is covered in different sections. The IVG is introduced in Section 3.1 and is extended as explained in Section 3.2. The greedy algorithm used to decide a variable mapping is presented in Section 3.3. Next, the computation of affinities and the assignment of latencies to memory instructions is covered in Section 3.4. A graphical example is shown in Section 3.5 to better understand the use of affinities. After that, instruction scheduling for cyclic code and for acyclic code are introduced in Section 3.6 and Section 3.7 respectively. Finally, the previous example is extended in Section 3.8.

3.1. The Instructions-to-Variables Graph (IVG)

The compiler uses the Instructions-to-Variables Graph (IVG) when mapping variables to address spaces. The IVG is a directed graph in which nodes represent memory instructions and variables of a program, and edges link instructions with variables. An edge between an instruction and a variable indicates that the instruction accesses that variable a certain amount of times (the weight of the edge). All this information is gathered through profiling and an example is shown in Figure 4.

The compiler assigns a mapping attribute to each program variable and to each variable node in the IVG in consequence. Such mapping attribute indicates the address space where the variable is bound to and has a value from the set *{0, 1}*. Variables mapped to the first address space will be assigned an address with the most significant bit set to one, while variables mapped to the second address space will be assigned an address with the most significant bit to zero.

Variable nodes in the IVG represent global, stack and heap variables. Each global variable of the program and each individual local variable in each routine become a node in the IVG. However, heap variables must be managed carefully since they are created, resized and freed at runtime. On one hand, heap variables should be distributed between the two address spaces

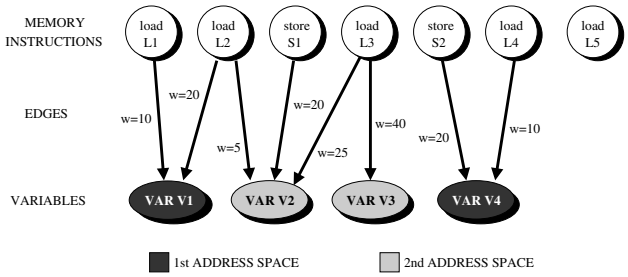

Figure 4. An example of an IVG. Note that each variable has been assigned to an address space.

at a fine granularity. This is translated into a rather large number of heap variables and gives more freedom to the compiler to distribute them with the best performance/energy consumption trade-off. On the other hand, heap variables must be managed so that such distribution is valid when the program is executed with different input sets. For example, we could assume that all heap variables must be mapped into the same address space. In this case, the computed distribution works with any input set. However, we may be losing opportunities to spread heap variables with different performance and energy characteristics between address spaces.

Our prior work showed that a good compromise between granularity and input independence (generality) is achieved by grouping together all heap variables created from the same dynamic call trace into the same IVG node [12]. A dynamic call trace consists of the run-time call trace at the point where *malloc* is called, avoiding cycles due to recursive function calls. For example, if function *main* calls function *foo*, that calls function *foo2*, that recursively calls itself and finally calls *malloc*, the dynamic call trace at this point consists of the tuple *{main, foo, foo2, malloc}*.

In such scenario, the distribution of heap variables is transparent to the user. Thus, there is a single *malloc* function and a single *free* function from the programmer's perspective. The compiler collects dynamic call traces during profiling and computes an address space for each of these traces. When *malloc* is called, the routine extracts the dynamic call trace at that point[1] and compares it with the traces collected during profiling. The memory allocation library then decides whether to allocate memory from one address space memory pool or another. There is no need for functions such as *malloc_first_addr_space*, *malloc_second_addr_space, free_first_addr_space*, and *free_second_addr_space*, which would make programming more complex and less portable.

3.2. Extending the IVG

The IVG is built with profiling information. Although profiling gives a good overview of the access pattern of memory instructions, there may still be missing information in the IVG. This may be translated into a considerable amount of remote accesses, which stall the processor and have an impact on performance. We have especially observed this phenomenon in benchmarks *gsmenc* and *mpeg2dec*.

One way to extend the IVG and augment it with information not observed during profiling consists of using the high-level name of variables. The name of the referenced variable is gathered for each memory instruction in the front-end of the compiler and it is propagated as instruction attributes to the back-end, where our mapping algorithm takes place. This information is used to add new edges in the IVG so that memory instructions are later scheduled in the correct cluster. For example, if load *L5* in Figure 4 is *load r1,@V3*, the high-level name of the referenced variable is *V3*, and we can add an edge connecting load *L5* to variable *V3*. The weight of the edge is not important in this case. Hence, load *L5* will be scheduled in cluster 2 if variable *V3* is mapped into the second address space.

This technique is only helpful when the name of the variable is not a pointer. For pointer accesses, an aggressive points-to analysis can be used to refine the IVG. Note that this points-to analysis can be aggressive, since it is not used for correctness, but for performance (avoid remote accesses).

We have applied these two techniques to benchmarks *gsmenc* and *mpeg2dec* and we have reduced the amount of remote accesses in these cases. For example, there were some instructions not executed during profiling in *mpeg2dec* that access global variables. In this case, their names were used to extend the IVG. On the other hand, a local variable is often passed by reference to a subroutine in *gsmenc*. This could be easily detected by a points-to analysis. However, we have extended the IVG by hand in this case. We believe it was not worth the effort to develop a full points-to analysis since this only affected a few instructions of one benchmark. Remote accesses results are shown in Section 4.2.

3.3. The Greedy Mapping Algorithm

The compiler is responsible for distributing variables between the two address spaces and generate code accordingly trying to maximize an objective function. The objective function takes into account performance and energy consumption at the same time. In this paper, we have used *energy·delay* (ED) and *energy·delay*^{2} (EDD) [5] as objective functions, and results using both are later shown in Section 4.

The mapping algorithm receives the IVG as input, along with the Data Dependence Graphs (DDGs) of all code regions of a program. Since our target processor has two clusters, a variable can be mapped into two different address spaces and each variable is assigned a mapping attribute from the set *{0, 1}*. The mapping algorithm starts by mapping all variables to the first address space. It then assigns the corresponding latency to memory instructions, computes the affinity of all instructions as explained later in Section 3.4, and schedules code as explained later in Section 3.6 and Section 3.7. After this initial assignment, the algorithm proceeds in a greedy manner trying to remap each variable from the first address space to the second one. In particular, for each variable, the algorithm computes the benefit

1. The dynamic instruction call trace is part of the function call convention. It implies the assignment of a unique *id* to each function and pushing/popping them from the stack along with their corresponding stack frames.

```
algorithm propagate_affinities
❶ set the affinity of all non-memory instructions to UNKNOWN
❷ slack=0
❸ max_slack = maximum slack of an edge in the DDG
❹ while slack <= max_slack
❺   build undirected graph DDG'={V,E'} from DDG={V,E}
          | (∀e'∈E' --> (e'∈E) and (slack(e)<=slack) and (type(e)=REG_FLOW))
❻   for each node v of DDG' | (affinity(v)=UNKNOWN) and (v is not a mem. inst)
❼     if there exists a path between v and a memory instruction in DDG'
          and this path does not contain any other memory instruction ; then
❽         affin(v)=average affin. of all mem. instructions rechable from v
      end if
     end for each
❾   slack=slack+1
   end while
end algorithm
```

Figure 5. Pseudo-code of the algorithm used to propagate affinities from memory instructions to the other instructions.

from remapping it to the second address space and remaps the variable with the best benefit if it is positive. Such benefit is computed using either ED or EDD depending on the experiments. Execution time (delay) is estimated for each variable by rescheduling the code regions that access that particular variable considering the new mapping. Energy, on the other hand, is estimated from the cache configuration and the schedule. A positive benefit is understood as a benefit that improves the current mapping configuration. The mapping algorithm iterates remapping one variable at a time from the first to the second address space until no more benefit is expected.

3.4. Computing Affinities and Assigning Latencies

Once variables are mapped into one of the two address spaces, memory instructions have a preferred cluster depending on the accessed variables. In order to describe this preference, we attach an affinity attribute to each memory instruction that will be later used to assign instructions to clusters. Such affinity is a value that ranges between 0 (the preferred cluster of the instruction is definitively cluster 1) and 1 (the preferred cluster of the instruction is definitively cluster 2) and is computed by the ratio between the number of accesses to variables mapped into the second address space and the total number of accesses. Since most memory instructions access a single variable [12], the affinity of most memory instructions is either 0 or 1. However, affinities between 0 and 1 are possible in case a memory instruction accesses variables mapped in different address spaces.

Once the affinity of memory instructions is computed, a latency is assigned to each one. Memory instructions with an affinity greater than 0.5 are assigned the latency of the cache module residing in cluster 2 because they will probably be scheduled in that cluster, whereas instructions with an affinity lower than or equal to 0.5 are assigned the latency of the cache module residing in cluster 1.

After computing the preferred cluster for each memory instruction, the preferred cluster of all other instructions is computed by propagating the affinity of memory instructions. The idea is to assign similar affinities to instructions that depend on the execution of the same memory instruction in order to avoid inter-cluster register communications. Since the execution of an instruction may depend on more than one memory instruction, affinities are combined.

In particular, affinities are computed by propagating the affinity of memory instructions to the other instructions only through register-flow dependences in the DDG. This is so because register-flow dependences are the only ones that require an inter-cluster register communication in case their source and target nodes are scheduled in different clusters. The rest of the edges are ignored in this step. In addition, the performance loss incurred by an inter-cluster register communication is often inversely proportional to the slack of its corresponding edge. Thus, the edge slacks are computed and affinities are propagated through the most critical edges first.

The slack of an edge is the number of cycles by which it can be stretched without increasing execution time. It is computed in a different manner for cyclic and acyclic code. Computing it for cyclic code is explained in Section 3.6, while it is explained in Section 3.7 for acyclic code.

The algorithm we have used to propagate affinities is shown in Figure 5. First, all non-memory instructions are assigned an unknown affinity and variables *max_slack* and *slack* are initialized to the largest edge slack in the graph and to zero respectively (lines ❶❷❸). After that, the algorithm iterates (line ❹) assigning affinities to instructions considering certain edges in each iteration. In particular, only register-flow dependences whose slack is lower than or equal to variable *slack* are considered in each iteration. Since variable *slack* was first initialized to zero and is increased in each iteration, affinities are propagated from most to least critical edges. In order to do so, the algorithm builds a subgraph with only the specified dependences (line ❺). This subgraph is transformed to an undirected graph in order to propagate affinities in any direction. For each node without a known affinity in the resulting subgraph (line ❻), its affinity is computed if there is a path between a memory instruction and it, and this path does not contain other memory instructions (line ❼). The affinity of the instruction is the arithmetic mean of the affinities of all reachable memory instructions avoiding paths with memory instructions within (line ❽). At the end of each iteration, variable *slack* is increased so that a "less critical" subgraph is built in the next iteration and affinities are propagated to nodes with an unknown affinity (line ❾).

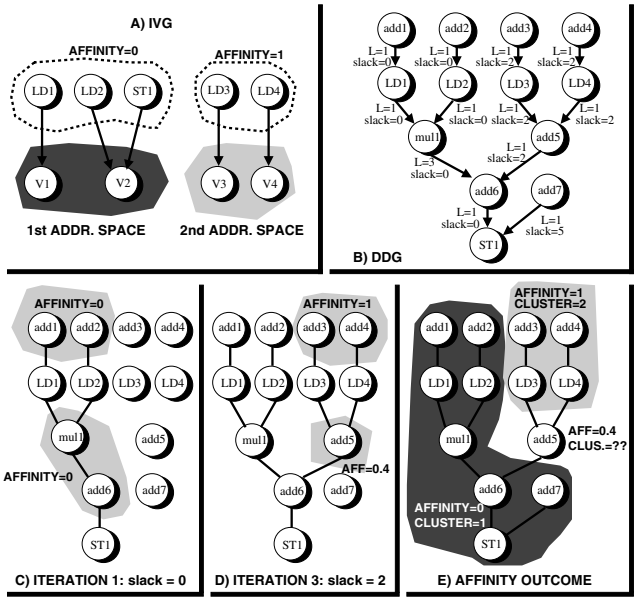

Figure 6. An example on how affinities are computed and propagated.

An example of how affinities are computed for a particular graph and used throughout instruction scheduling is presented next.

3.5. Example of Affinity Use

Let us use a simple example to show how affinities are computed and used. Imagine the scenario shown in Figure 6 where variables *V1* and *V2* have been mapped into the first address space and variables *V3* and *V4* have been mapped into the second address space as shown in the IVG in Figure 6(A). In addition, the DDG of an acyclic code region is depicted in Figure 6(B) where only register-flow dependences have been considered. For simplicity we assume that the latency (L) of all instructions is 1 cycle, except for the multiplication, whose latency is 3 cycles. Note that we assume that the latency of both cache modules is the same.

First, the affinity of memory instructions is computed. Since each memory instruction accesses a single variable, their affinities are binary. Instructions *LD1*, *LD2* and *ST1* have an affinity of 0, and instructions *LD3* and *LD4* have an affinity of 1 as shown in Figure 6(A). Thus, *LD1*, *LD2* and *ST1* are assigned the latency of the first address space whereas *LD3* and *LD4* are assigned the latency of the second address space, which is 1 cycle in both cases. Once latencies have been assigned, the edge slacks are computed and are pictured close to each edge in Figure 6(B). In this case, since the code region is acyclic, all slacks are length slacks as later explained in Section 3.7.

Next, affinities are propagated. Only edges with a slack of zero are considered in the first iteration, leading to a subgraph shown in Figure 6(C). The resulting subgraph is transformed into an undirected graph structure. At this point, the affinity of instructions *add1*, *add2*, *mul1* and *add6* is computed. Since all reachable memory instructions in this subgraph have an affinity of 0, the affinity of these instructions is 0 as well. In the third iteration, edges whose slack is lower than or equal to two cycles are considered, and the algorithm computes the affinity of instructions *add3*, *add4* and *add5*. The resulting subgraph is shown in Figure 6(D). Note that the affinity computed previously for other instructions is not recomputed. Although there exists a path between *LD1* and *add3* in the undirected subgraph, the affinity of *LD1* is not considered to compute *add3*'s because the path includes another memory instruction (*LD3*). Hence the only path considered for *add3* is the path consisting of *{LD3, add3}*, while the path *{LD4, add4}* is used for instruction *add4*. Both of their affinities are 1. If all the paths were considered, for example, the affinity of instruction *add3* would be 0.4 which is not intuitive since *add3* is very tied to *LD3*, whose affinity is 1. On the other hand, the affinity of instruction *add5* is 0.4. This is the arithmetic mean of the affinities of memory instructions reachable from *add5* in the subgraph, excluding paths containing other memory instructions. These reachable memory instructions include *LD1*, *LD2*, *LD3*, *LD4* and *ST1*.

Finally, the algorithm finishes propagating the affinities after the sixth iteration, when variable *slack* is equal to the maximum edge slack, which is five cycles. The result of the propagation is shown in Figure 6(E).

3.6. Instruction Scheduling for Cyclic Code

Modulo scheduling is an effective technique to extract instruction-level parallelism (ILP) from loops by overlapping the execution of successive iterations of the original loop without the need to unroll it [7]. The parameter that most affects the performance of a modulo scheduled loop is the Initiation Interval (II), which is the number of cycles between the initiation of consecutive iterations. Modulo scheduling is a well-understood technique used by many current compilers. In this work, modulo scheduling has been applied to innermost loops that iterate at least 8 times during profiling.

Edges in a Data Dependence Graph (DDG) of a modulo scheduled loop have two kinds of slack: the recurrence slack and the length slack. The former represents the number of cycles that the edge can be stretched without increasing the II, whereas the latter is the number of cycles that the edge can be stretched without increasing the schedule length. In order to combine both slacks into a single slack value, the smallest value is used for each edge (the most restrictive slack). Note that instructions that do not belong to any recurrence have an infinite recurrence slack and their slack corresponds to their length slack.

Nodes in the DDG are ordered using the Swing Modulo Scheduling (SMS) heuristic [19]. SMS is one of the most effective modulo scheduling techniques [8]. Once the nodes are ordered, the algorithm proceeds by scheduling one instruction at a time. For each instruction, the set of possible clusters where it can be scheduled is computed. This set contains the clusters with enough free resources to execute the instruction. If the instruction cannot be scheduled in any cluster, the II is increased and instruction scheduling starts again.

On the other hand, if the set of possible clusters is not empty, it is ordered using the following criterion. First, if the instruction has a "strong" preferred cluster which belongs to the set of possible clusters, it is added at the beginning of the set so that it will be probed first. For instructions without a strong preferred cluster, the set of possible clusters is ordered so that clusters where inter-cluster communications are minimized and workload balance is maximized are selected first. An affinity range is used to define when a preferred cluster is a "strong" preferred cluster. For instance, an affinity range of *(0.1, 0.9)* specifies that cluster 1 is the "strong" preferred cluster when the affinity of the instruction is lower than or equal to 0.1, while cluster 2 is the "strong" preferred cluster when the affinity is greater than or equal to 0.9. Instructions with an affinity between 0.1 and 0.9 are considered not to have a strong preferred cluster. An affinity range analysis was performed in order to choose a proper range and we found that the range *(0,1)* was the best one [13]. Hence, we have used such a range throughout the rest of the paper. Finally, the instruction is scheduled in the first cluster of the set where a valid slot is found. If the schedule is not possible, the II is increased and instruction scheduling starts again.

3.7. Instruction Scheduling for Acyclic Code

Acyclic code regions are scheduled using list scheduling. In this work, these regions include: innermost loops with function calls within, innermost loops that iterate less than 8 times during profiling, and hyperblocks and basic blocks not in innermost loops.

The affinity of memory instructions is computed and propagated to the rest of the instructions as explained in Section 3.4. However, in this case, the slack of an edge is only restricted by the length of the schedule and not by recurrences. In the process of scheduling each instruction, priority is given to the strong preferred cluster (if any) as was explained in Section 3.6.

3.8. Example Continuation

Recalling the example shown in Figure 6, the instruction scheduler will try to schedule instructions with an affinity of 0 in cluster 1 and instructions with an affinity of 1 in cluster 2. Instruction *add5* will be scheduled in one cluster or the other depending on the affinity range. Since we found that the best range is *(0,1)*, *add5* does not have a strong preferred cluster and it is scheduled in the cluster where register communications are minimized and workload balance maximized. In this case, *add5* will be assigned to cluster 2.

4. Evaluation

In this section, the proposed cache configurations are evaluated. The evaluation framework is presented in Section 4.1. Next, remote accesses are quantified in Section 4.2. After that, EDD and ED results are presented in Section 4.3. One of the main conclusions of these results is that there is no single distributed configuration that is the best one for all programs. Thus, we also evaluate a reconfigurable cache scheme in Section 4.4.

4.1. Evaluation Framework

The IMPACT compiler [6] has been used to compile the benchmarks and optimize them. The benchmarks we have used are the Mediabench suite [17] because they represent typical workloads to be executed in media or embedded processors such as DSPs. The benchmarks and their inputs are summarized in Table 1. All benchmarks have been simulated until completion.

Different distributed cache configurations have been evaluated: FAST+NONE, FAST+FAST, FAST+SLOW, SLOW+SLOW, and SLOW+NONE. They have been compared to a clustered architecture with a unified data cache in terms of *energy·delay* (ED) and *energy·delay*^{2} (EDD). In each case, the processor consists of two clusters and each has one integer, one memory and one floating point functional unit. The address of memory references is computed in the memory functional unit. The latency of a fast cache module is 2 cycles, while the latency of a slow module is 4 cycles.

	Profile data set	Execution data set
adpcmdec	clinton.adpcm	S_16_44.adpcm
adpcmenc	clinton.pcm	S_16_44.pcm
epicdec	test_image.pgm.E	titanic3.pgm.E
epicenc	test_image.pgm	titanic3.pgm
g721dec	clinton.g721	S_16_44.g721
g721enc	clinton.pcm	S_16_44.pcm
gsmdec	clint.pcm.run.gsm	S_16_44.pcm.gsm
gsmenc	clinton.pcm	S_16_44.pcm
jpegdec	testimg.jpg	monalisa.jpg
jpegenc	testimg.ppm	monalisa.ppm
mpeg2dec	mei16v2.m2v	tek6.m2v
pegwitdec	pegwit.enc	tech_rep.txt.enc
pegwitenc	pgptest.plain	tech_rep.txt
pgpdec	pgptext.pgp	tech_rep.txt.enc
pgpenc	pgptest.plain	tech_rep.txt
rasta	ex5_c1.wav	ex5_c1.wav

Table 1. Benchmarks and inputs used in simulations.

In the case of a clustered processor with a unified cache, we have assumed that an extra delay is incurred to access the cache because it cannot be close to both clusters. Two delay values have been used: 1 cycle (half cycle to send the request to the cache, plus half cycle to receive the reply) and 2 cycles. In addition, we use a banked cache in this case so that each bank has the same port and latency configuration as a module in the clustered cache scheme. A banked cache configuration where each bank has 1 read/write port is more energy efficient than a monolithic cache with two read/write ports [24][12]. The banks are either fast banks or slow banks. These two bank configurations together with the two delay overheads result in four different

unified cache schemes. These are: a fast unified scheme with a 3-cycle latency (half cycle to send the request to the cache, 2 cycles to access the cache and half cycle to send back the reply), a fast unified scheme with a 4-cycle latency (one cycle to send the request to the cache, 2 cycles to access the cache and one cycle to send back the reply), and slow unified schemes with latencies of 5 and 6 cycles. Furthermore, we have used state-of-the-art instruction scheduling techniques to generate code for such architecture [2]. These techniques basically consist of computing a partitioning of the Data Dependence Graph in order to guide the assignment of instructions to clusters.

Each cache module or bank is 2-way set-associative and has a size of 4KB with 32-byte blocks. Hence, the total cache size for the FAST+NONE and SLOW+NONE is 4KB, while the total size for the rest distributed schemes and the unified cache schemes is 8KB.

We have assumed that the cache consumes 1/3 of the processor energy and that leakage accounts for 50% of the total energy, which is consistent with trends shown in [15][22][26]. ED and EDD values reported in the following sections specify the trade-off between performance and energy consumption in the whole processor and not only in the cache. Such values are computed with respect to the same baseline so that numbers can be compared directly. For simplicity, we have chosen the configuration FAST+NONE to be the baseline architecture. Hence, a configuration with an ED of 0.9 is 10% better in ED than the FAST+NONE configuration, whereas a configuration with an ED of 1.1 is 10% worse than this baseline.

Finally, we use two non-pipelined buses to communicate clusters with a latency of 2 cycles. The energy consumed by inter-cluster communications cannot be computed easily without a floor plant of the processor since wire widths and distances between clusters are unknown. Hence, we have simulated three different energy scenarios. The first one assumes that a register communication instruction consumes the same energy as any other generic instruction in the processor. The other two scenarios assume that a register communication instruction consumes twice and four times the energy of any other generic instruction. We refer these cases to as W=1, W=2 and W=4 respectively, which can be understood as the energy weight of communication instructions with respect to the rest. Since they account for around 15% of the total number of dynamic instructions, these three scenarios correspond approximately to situations in which inter-cluster communications consume 15% of the processor energy (excluding the cache) for W=1, 26% for W=2 (*0.15*2 / (0.85+0.15*2)*), and 41% for W=4 (*0.15*4 / (0.85 + 0.15*4)*).

4.2. Remote Accesses

Remote accesses occur when one instruction accesses a datum mapped in the cache module of the other cluster. In Table 2, we show the ratio of remote accesses for each benchmark over one thousand memory accesses (‰) using the FAST+SLOW scheme and W=2. In parenthesis we show the same ratio for benchmarks *gsmenc* and *mpeg2dec* before extending the IVG as explained in Section 3.2. As it can be seen, the reduction in remote accesses is big in these two cases. We only chose these two benchmarks because they were the ones with a larger impact on performance due to remote accesses. In particular, stall time was reduced from 3.7M cycles to 74 cycles in *gsmenc*, leading to an overall execution time reduction of 3.6%, whereas stall time was reduced by 42x in *mpeg2dec*, leading to an overall execution time reduction close to 2%. In summary, remote accesses are infrequent for all benchmarks and could be reduced further by doing a more exhaustive extension of their respective IVGs. Remote accesses are also very infrequent in the FAST+FAST and SLOW+SLOW schemes.

The impact of remote accesses into execution time is also shown in Table 2. These results show the proportion of time that the processor is stalled performing a remote access over total execution time. This is expressed as per thousand (‰). As can be observed, the impact of remote accesses is proportional to their amount and is negligible for all benchmarks.

	Ratio(‰)	Exec(‰)		Ratio(‰)	Exec(‰)
adpcmdec	0.2	0.3	**jpegdec**	0.2	1.4
adpcmenc	3.4	2.6	**jpegenc**	0	0
epicdec	0	0	**mpeg2dec**	0.1 (13.2)	0.3
epicenc	3.1	5.8	**pegwitdec**	0.2	0.4
g721dec	0	0	**pegwitenc**	5.5	11.9
g721enc	0	0	**pgpdec**	1.7	7.6
gsmdec	2.1	2.9	**pgpenc**	3.7	11.4
gsmenc	0 (34.2)	0	**rasta**	0	0

Table 2. Ratio of remote accesses and overall execution time due to them.

4.3. EDD and ED Results

In Figure 7, results are shown for each benchmark with EDD as the objective function and W=2. The top graph plots execution time for the five distributed cache schemes: FAST+NONE, FAST+FAST, FAST+SLOW, SLOW+SLOW and SLOW+NONE. Execution time is normalized to that of the FAST+NONE scheme which has been used as the baseline configuration. The middle graph in Figure 7 shows the distribution of memory accesses between the two cache modules for the FAST+FAST, FAST+SLOW and SLOW+SLOW schemes. Memory access results for the FAST+NONE and the SLOW+NONE schemes are not shown since they only have a single cache module. In the case of the FAST+SLOW configuration, the white portion of the bar represents accesses to the slow cache module. Note that around 75% of the memory accesses are concentrated in the first cache module. This is explained by the fact that everything is mapped into the first address space by default and data are moved to the other space when benefit is observed. With the FAST+SLOW organization, moving a variable to the slow address space saves energy and permits a better usage of memory ports and cluster resources, but at an expense in latency increase and in inter-cluster communications. Moving a variable in the case of the FAST+FAST

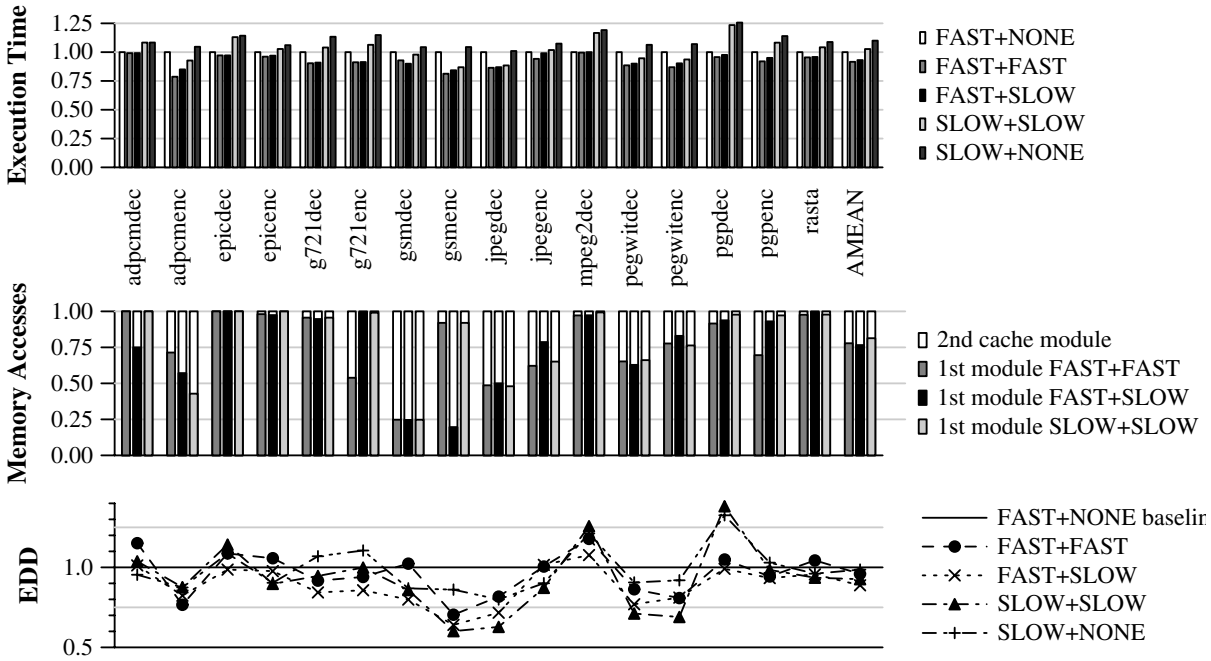

Figure 7. Execution time, memory accesses and *energy·delay*^{2} (EDD) results with W=2.

and SLOW+SLOW schemes only implies a better use of memory ports and cluster resources, at an expense in inter-cluster communications.

Finally, the bottom graph in Figure 7 shows EDD results for all configurations with respect to the FAST+NONE baseline. On average, the FAST+SLOW scheme is the best one in the trade-off between performance and energy consumption. In particular, the FAST+SLOW organization is 11% better in EDD than the FAST+NONE scheme. In addition, it is 4% better than the SLOW+SLOW approach, which is the second best scheme. Furthermore, the results for the FAST+SLOW scheme are more stable than those of SLOW+SLOW. For instance, the SLOW+SLOW scheme works very well for benchmarks *jpegdec*, *pegwitdec* and *pegwitenc* compared to the other configurations. However, it is a bad configuration for *epicdec*, *mpeg2dec* and *pgpdec*, where EDD is 1.14, 1.26 and 1.38 that of the baseline configuration.

One important conclusion that can be extracted from Figure 7 is that there is not a single configuration that is the best for all benchmarks. The FAST+FAST configuration turns out to be the most appropriate one when the benchmark is sensitive to memory latency and the number of memory ports, as is the case for *adpcmenc*. In those cases where the programs are sensitive to latency but insensitive to the number of memory ports, the FAST+NONE scheme works very well. An example is the *mpeg2dec* benchmark. In addition, when a benchmark is sensitive to the number of memory ports, but little sensitive to memory latency, the SLOW+SLOW scheme outperforms the others. Programs *gsmenc*, *jpegdec*, *pegwitdec* and *pegwitenc* are good examples of this latter group. Finally, the best scheme for benchmarks that are insensitive to the number of ports and memory latency is the SLOW+NONE. Although *adpcmdec* is slightly sensitive to memory latency, the benefits of having a single slow power-aware cache module overcomes the performance loss due to the restrictions mentioned above. Thus, *adpcmdec* achieves the best results using the SLOW+NONE configuration. The FAST+SLOW scheme falls in between all other schemes, achieves a compromise between port and latency sensitivity, and between performance and energy consumption, and is the best scheme on average.

	FAST+ NONE	FAST+ FAST	FAST+ SLOW	SLOW+ SLOW	SLOW+ NONE
W=1	1	0.95	0.88	0.92	0.99
W=2	1	0.96	0.89	0.93	0.99
W=4	1	0.98	0.90	0.93	0.98

Table 3. *Energy·delay*^{2} (EDD) results for different W factors.

Results are similar with other weight factors. These results are shown in Table 3. Overall, the FAST+SLOW scheme outperforms the FAST+NONE scheme by 12%, 11% and 10% with W=1, W=2 and W=4 respectively and it is 4.2%, 4.1% and 3.8% better than the SLOW+SLOW scheme with W=1, W=2 and W=4 respectively, which is the second best scheme. However, trends indicate that results get closer to the baseline architecture as more energy weight is given to inter-cluster register communications because everything tends to be scheduled in one cluster. In this situation, it is more efficient to have a scheme with a single cache module. We have simulated a case in which a weight of 16 instructions is assigned to inter-cluster communications, so they consume around 74% of the processor energy excluding the cache. In this case, the 12% EDD benefit of FAST+SLOW compared to FAST+NONE with W=1 is trans-

lated into a benefit of 7% with W=16, and the amount of fast memory accesses is increased from 75% with W=1 to 84% with W=16.

All distributed schemes have better EDD results than any of the schemes with a unified cache. Results with W=2 are summarized for all configurations in Table 4, where four results are plot for a processor with a unified data cache depending on the cache latency and the delay incurred to access it. For instance, the EDD of a fast unified cache scheme is 1.14 and 1.29 that of the baseline architecture with a delay overhead of 1 and 2 cycles respectively, while it is 1.10 and 1.25 for a slow unified cache scheme depending on the delay overhead. Thus, the best distributed configuration (FAST+SLOW) is 22%-31% better in EDD than a fast unified organization and 19%-29% better than a slow unified configuration.

	F+N	F+F	F+S	S+S	S+N	UNIFIED FAST		UNIFIED SLOW	
						L=3	L=4	L=5	L=6
EDD	1	0.96	0.89	0.93	0.99	1.14	1.29	1.10	1.25
ED	1	1.04	0.94	0.89	0.89	1.16	1.25	1.00	1.07

Table 4. Average results with W=2.

Lastly, results when ED is used as the objective function with W=2 are summarized for all configurations in Table 4. In this case, the SLOW+SLOW configuration turns out to be the best one when ED is used instead of EDD. In particular, the SLOW+SLOW scheme outperforms the baseline architecture by 12%, 11% and 12% with W=1, W=2 and W=4 respectively. The second best configuration is the SLOW+NONE which outperforms the baseline by 10%, 11% and 11% with W=1, W=2 and W=4, while the FAST+SLOW scheme outperforms the baseline by 7%, 6% and 4% with W=1, W=2 and W=4. It is not surprising that slow configurations are better with ED than with EDD because in the former energy consumption is more important relative to execution time. In this case, the best distributed cache configuration (SLOW+SLOW) is 24%-29% better than a fast unified organization and 11%-19% better than a slow unified scheme.

4.4. Results for a Reconfigurable Cache

Based on the observation that there is not a cache configuration that is the best for all benchmarks, we have also evaluated a scheme in which the cache may be configured on an application basis. In this case, each cache module can operate with three different modes and the operating system is responsible to configure it depending on the application that is running. The three modes are: turn off the cache module, put the cache module into fast mode, and put the cache module into slow mode. For the last two modes, we must set the supply and threshold voltages accordingly. Thus, we can choose a configuration among FAST+NONE, FAST+FAST, FAST+SLOW, SLOW+SLOW and SLOW+NONE for a given application. To enable such reconfigurable cache architecture we need two different supply and threshold voltages for the cache. This is similar to drowsy caches [10] but with less complexity, since we do not allow different voltages for different cache lines. In this case, the voltage is the same for the entire cache module.

The compiler statically computes the best configuration for a given application, schedules code accordingly and reflects this information in the binary file. We have used a simple technique to compute the best configuration, which consists of scheduling the benchmark for all five possible configurations and choose the best one in terms of expected ED or EDD. A more sophisticated approach could be used but we wanted to evaluate the potential of the reconfigurable cache scheme. On a context switch, the operating system may decide to reconfigure the cache depending on the process being scheduled out, and the process being scheduled in. Based on previous work [10], the overhead of reconfiguring the cache is 1 or 2 cycles and thus, it is negligible.

	Best scheme for non-configurable cache	Reconfigurable cache scheme
average EDD	0.888 (**FAST+SLOW**)	0.856
average ED	0.886 (**SLOW+SLOW**)	0.860

Table 5. Average *energy·delay*^{2} (EDD) and *energy·delay* (ED) values for a reconfigurable heterogeneous cache.

In Table 5 average EDD and ED values are shown for this reconfigurable scheme. The first column shows the results presented in previous sections for a non-reconfigurable cache scheme. In each case, the result for the best scheme is shown (the FAST+SLOW and the SLOW+SLOW schemes for EDD and ED respectively). The second column shows the results for the reconfigurable cache scheme. We can see that results can still be improved by 3-4% when using reconfiguration depending on the objective function.

5. Related Work

Heterogeneity has been exploited in the functional units [23][29] and in the memory hierarchy [1]. In the latter, the authors divide the data cache into a fast and a slow module for an out-of-order processor. In this paper, we explore memory configurations for a clustered VLIW processor. Hence, the proposed techniques are radically different than those in [1]. Furthermore, the authors of [1] conclude that one of their proposed schemes is not very energy efficient.

Regarding the embedded domain, in which VLIW processors are common, several works explore the use of a scratch-pad memory ([20][4][3][28], among others) because they consume little energy. However, the use of scratch-pad memories is orthogonal to the use of a cache hierarchy. In addition, most high-performance embedded processors include some kind of memory hierarchy with at least one level of instruction and data caches ([15][25][9][27]).

Multi-module caches have also been proposed as a solution to either better exploit data locality [14] or reduce *energy·delay* related functions [18][16]. However, none of the proposals targets clustered processors.

Finally, some works explore the use of distributed cache configurations for clustered VLIW processors [21][11]. In this case, the goal of the proposals is performance and no attention is paid to energy consumption.

6. Conclusions

In this paper we have evaluated an architectural scheme that consists of dividing the cache into two modules and assigning each module to one cluster in a VLIW processor with two clusters. The cache modules can be set up as fast power-hungry modules or slow power-aware modules, leading to several cache configurations. Furthermore, we have developed compiler techniques to exploit the underlying cache configuration efficiently.

With EDD, the best scheme is an heterogeneous one, where the cache consists of a fast module and a slow module each one assigned to a different cluster. This scheme outperforms traditional cache organizations by an average of 19%-31% in EDD depending on its latency. For example, when compared to a conventional unified cache configured as a fast cache, the proposed distributed scheme reduces dynamic energy by 8%, leakage by 20%, and execution time by 4%, resulting in a 21% gain in EDD. On the other hand, when ED is used instead, the best distributed configuration outperforms traditional cache organizations by 11%-29%.

We have also observed that there is not a single cache configuration that is the best for all benchmarks. Thus, we have also explored an approach in which the cache can be reconfigured on a context switch. Results demonstrate that a reconfigurable distributed cache is 3%-4% better than any of the proposed non-reconfigurable distributed organizations and 14%-34% better than any of the traditional unified cache organizations.

Acknowledgments

This work has been partially supported by the Spanish Ministry of Education and Science under contract TIN2004-03072, Feder funds, a grant *UPC per la Recerca* and Intel.

References

[1] J. Abella, A. González, “Power Efficient Data Cache Designs”, in *Procs. of 21st Int. Conf. on Computer Design*, pp. 8-13, Oct. 2003

[2] A. Aletà, J. M. Codina, J. Sánchez, A. González, and D. R. Kaeli, "Exploiting Pseudo-Schedules to Guide Data Dependence Graph Partitioning", in *Procs. of the Int. Conf. on Parallel Architectures and Compilation Techniques*, pp. 281-290, 2002

[3] F. Angiolini, L. Benini, A. Caprara, “Polynomial-Time Algorithm for On-Chip Scratchpad Memory Partitioning”, in *Procs. of Int. Conf. on Compilers, Architectures, and Synthesis for Embedded Systems (CASES)*, Oct.-Nov. 2003

[4] O. Avissar, R. Barua, D. Stewart, “Heterogeneous Memory Management for Embedded Systems”, in *Procs. of Int. Conf. on Compilers, Architectures, and Synthesis for Embedded Systems (CASES)*, Nov. 2001

[5] D. M. Brooks, P. W. Cook, P. Bose, S. E. Schuster, H. Jacobson, P. N. Kudva, A. Buyuktosunoglu, J. Wellman, V. Zyuban, M. Gupta, “Power-Aware Microarchitecture: Design and Modeling Challenges for Next-Generation Microprocessors”, in *IEEE Micro Volume 20, Issue 6*, Nov. 2000

[6] P.P. Chang, S.A. Mahlke, W.Y. Chen, N.J. Water, and W.W. Hwu, "IMPACT: An Architectural Framework for Multiple-Instruction-Issue Processors", in *Procs. of the 18th Int. Symp. on Computer Architecture*, pp. 266-275, May 1991

[7] A. Charlesworth, “An Approach to Scientific Array Processing: The Architectural Design of the AP120B/FPS-164 Family”, in *Computer*, 14(9), pp.18-27, 1981

[8] J. M. Codina, J. Llosa and A. González, “A Comparative Study of Modulo Scheduling Techniques”, in *Procs. of Int. Conference on Supercomputing*, June 2002

[9] P. Faraboschi, G. Brown, J. Fisher, G. Desoli, F. Homewood, “Lx: A Technology Platform for Customizable VLIW Embedded Processing”, in *Procs. of 27th Annual Int. Symp. on Computer Architecture*, 2000

[10] K. Flautner, N. S. Kim, S. Martin, D. Blaauw, and T. Mudge, “Drowsy Caches: Simple Techniques for Reducing Leakage Power”, in *Procs. of 29th Annual Int. Symp. on Computer Architecture*, 2002

[11] E. Gibert, J. Sánchez, A. González, "Flexible Compiler-Managed L0 Buffers for Clustered VLIW Processors", in *Procs. of 36th Int. Symp. on Microarchitecture*, pp. 315-325, Dec. 2003

[12] E. Gibert, J. Abella, J. Sánchez, A. González, X. Vera, “An Energy-Effective Variable-Based Multi-Module L1 Data Cache for VLIW Processors”, *technical report UPC-DAC-RR-ARCO-2004-4,* Universitat Politècnica de Catalunya (UPC), July 2004

[13] E. Gibert, J. Abella, J. Sánchez, X. Vera, A. González, “Variable-Based Multi-Module Data Caches for Clustered VLIW Processors”, *technical report UPC-DAC-RR-ARCO-2005-2,* Universitat Politècnica de Catalunya (UPC), March 2005

[14] A. González, C. Aliagas, M. Valero, “A Data Cache with Multiple Caching Strategies Tuned to Different Types of Locality”, in *Procs. of Int. Conf. on Supercomputing*, July 1995

[15] S. Hill, “The ARM10 Family of Advanced Embedded Microprocessor Cores”, in *Procs. of Hot Chips 13*, Aug. 2001.

[16] M. Huang, J. Renau, S. Yoo, J. Torrellas, “L1 Data Cache Decomposition for Energy Efficiency”, in *Procs. of Int. Symp. On Low Power Electronics and Design*, Aug. 2001

[17] C. Lee, M. Potkonjak, and W.H. Mangione-Smith, “MediaBench: a Tool for Evaluating and Synthesizing Multimedia and Communication Systems”, in *Procs. of 30th Int. Symp. on Microarchitecture*, pp. 330-335, Dec. 1997

[18] H. Lee, G. Tyson, “Region-Based Caching: An Energy-Delay Efficient Memory Architecture for Embedded Processors”, in *Procs. of Int. Conf. on Compilers, Architectures, and Synthesis for Embedded Systems (CASES)*, Nov. 2000

[19] J. Llosa, A. González, E. Ayguadé and M. Valero, “Swing Modulo Scheduling”, in *Procs. of Int. Conf. on Parallel Architectures and Compilation Techniques*, pp.80-86, Oct. 1996

[20] P. Panda, N. Dutt, A. Nicolau, “Efficient Utilization of Scratch-Pad Memory in Embedded Processor Applications”, in *Procs. of European Design and Test Conference*, March 1997

[21] J. Sánchez, and A. González, “Modulo Scheduling for a Fully-Distributed Clustered VLIW Architecture”, in *Procs. of 33rd Int. Symp. on Microarchitecture*, Dec. 2000

[22] Semiconductor Industry Association (SIA), “International Technology Roadmap for Semiconductors 2001”, *available at http://public/itrs.net/files/2001ITRS*

[23] J. Seng, E. Tune, D. Tullsen, “Reducing Power with Dynamic Critical Path Information”, in *Procs. of 34th Int. Symp. on Microarchitecture*, Dec. 2001

[24] P. Shivakumar, N. Jouppi, “CACTI 3.0: An Integrated Cache Timing, Power, and Area Model”, *Western Research Lab technical report 2001/2*, Dec. 2001

[25] G. Slavenburg, S. Rathnam, H. Dijkstra, “The TriMedia TM-1 PCI VLIW Media Processor”, in *Procs. of Hot Chips 8*, Aug. 1996

[26] Synopsis Inc., “Managing Power in Ultra Deep Submicron ASIC/IC Design”, *Synopsis white paper*, May 2002

[27] Texas Instruments Inc., “TMS320C6000[tm] Platform Overview”, *available at http://dspvillage.ti.com*

[28] S. Udayakumaran, R. Barua, “Compiler-Decided Dynamic Memory Allocation for Scratch-Pad Based Embedded Systems”, *Procs. of Int. Conf. on Compilers, Architectures, and Synthesis for Embedded Systems (CASES)*, Nov. 2003

[29] W. Zhang, N. Vijaykrishnan, M. Kandemir, M.J. Irwin, D. Duarte, Y-F. Tsai, “Exploiting VLIW Schedule Slacks for Dynamic and Leakage Energy Reduction”, in *Procs. of 34th Int. Symp. on Microarchitecture*, Dec. 2001