

Increasing Power Efficiency of Multi-Core Network Processors through Data Filtering

Gokhan Memik

Department of Electrical Engineering
University of California, Los Angeles
++1-(213) 509 6792
memik@ee.ucla.edu

William H. Mangione-Smith

Department of Electrical Engineering
University of California, Los Angeles
++1-(310) 206 4195
billms@ee.ucla.edu

ABSTRACT

We propose and evaluate a data filtering method to reduce the power consumption of high-end processors with multiple execution cores. Although the proposed method can be applied to a wide variety of multi-processor systems including MPPs, SMPs and any type of single-chip multiprocessor, we concentrate on Network Processors. The proposed method uses an execution unit called Data Filtering Engine that processes data with low temporal locality before it is placed on the system bus. The execution cores use locality to decide which load instructions have low temporal locality and which portion of the surrounding code should be off-loaded to the data filtering engine.

Our technique reduces the power consumption, because a) the low temporal data is processed on the data filtering engine before it is placed onto the high capacitance system bus, and b) the conflict misses caused by low temporal data are reduced resulting in fewer accesses to the L2 cache. Specifically, we show that our technique reduces the bus accesses in representative applications by as much as 46.8% (26.5% on average) and reduces the overall power by as much as 15.6% (8.6% on average) on a single-core processor. It also improves the performance by as much as 76.7% (29.7% on average) for a processor with 16 execution cores.

Categories and Subject Descriptors

B.3 [Memory Structures]: Design Styles - Shared Memory; B.7 [Integrated Circuits]: Types and Design Styles - Memory Technologies; C.1 [Processor Architectures]: Parallel Architectures.

General Terms

Design, Measurement, Performance

Keywords

Network processors, data locality, power reduction, chip multiprocessors, remote procedure call.

1. INTRODUCTION

Power has been traditionally a limited resource and one of the most important design criteria for mobile processors and embedded systems. Due in part to increased logic density, power dissipation in high performance processors is also becoming a major design factor. The increased number of transistors causes processors to dissipate more heat, which in return reduces processor performance and reliability.

Network Processors (NPs) are processors optimized for networking applications. Until recently, processing elements in the networks were either general-purpose processors or ASIC designs. Since

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

CASES 2002, October 8–11, 2002, Grenoble, France.
Copyright 2002 ACM 1-58113-575-0/02/0010...\$5.00.

general-purpose processors are software programmable, they are very flexible in implementing different networking tasks. ASICs, on the other hand, typically have better performance. However, if there is a change in the protocol or the application, it is hard to reflect the change in the design. With the increasing number of new protocols and increasing link speeds, there is a need for processing elements that satisfy the processing and flexibility requirements of the modern networks. NPs fill this gap by combining network specific processing elements with software programmability.

In this paper, we present an architectural technique to reduce the power consumption of multi-core processors. Specifically, we:

- present simulation results showing that most of the misses in networking applications are caused by only a few instructions,
- devise a power reduction technique, which is a combination of a locality detection mechanism and an execution engine,
- discuss a fine-grain technique to offload code segments to the engine, and
- conduct simulation experiments to evaluate the effectiveness of our technique.

Although our technique can be efficiently employed by a variety of multi-processor systems, we concentrate on NPUs, because most of the NPUs follow the single-chip multiprocessor design methodology [18] (e.g. Intel IXP 1200 [11] (7 execution cores) and IBM PowerNP [12] (17 execution cores)). In addition, these chips consume significant power: IBM PowerNP [12] consumes 20W during typical operations, whereas C-Port C-5 [7] consumes 15W. The multiple execution cores are often connected by a global system bus as shown in Figure 1. The capacitive load on the processor's input/output drivers is usually much larger (by orders of magnitude) than that on the internal nodes of the processor [23]. Consequently, a significant portion of the power is consumed by the bus.

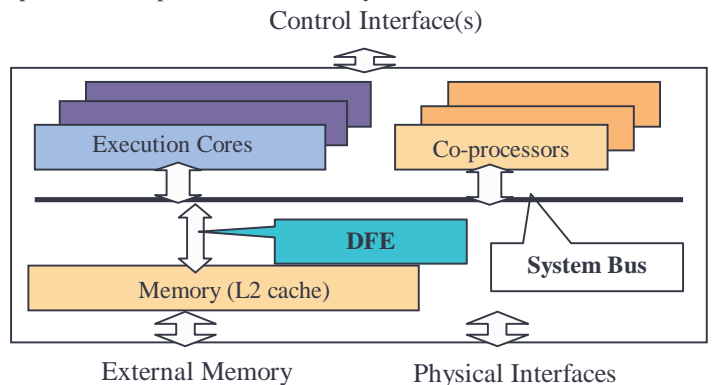


Figure 1. A generic Network Processor design and the location of the proposed DFE

Our technique uses two structures to achieve desired reduction in power consumption. First, the shared global memory is connected to the system bus through an execution unit named Data Filtering Engine (DFE). If not activated, the DFE passes the data to the bus, and hence has no effect on the execution of the processors. If the DFE is activated by an execution core, it processes the data and places the results on the bus. The goal is to process the low-temporal data within the DFE so that the number of bus accesses is reduced. By reducing the bus accesses and the cache misses caused by these low-temporal accesses, our technique achieves significant power reduction in the processor. Second, the low temporal accesses are determined by the execution cores using a locality prediction table (LPT). This table stores information from prior loads. The LPT stores the program counter (PC) of the instruction as well as the miss/hit behavior in the recent executions of the instruction. The LPT is also used to determine which section of the code should be offloaded to the DFE. The details of the LPT will be explained in Section 2.1. In the following subsection, we present simulation numbers motivating the use of data filtering for power reduction.

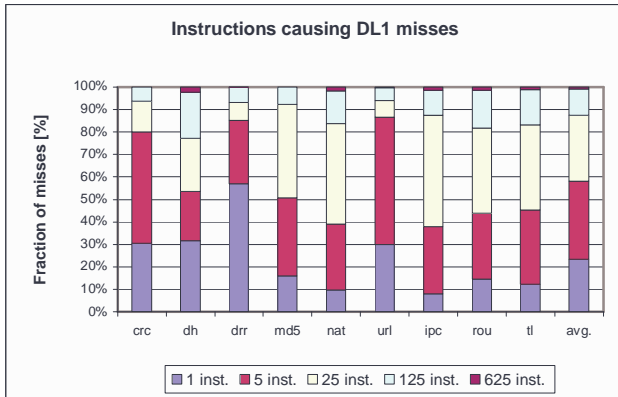


Figure 2. Number of instructions causing the DL1 misses

1.1 Motivation

In many applications, the majority of misses in the L1 data cache are caused by a few instructions [1]. We have performed a set of simulations to see the cache miss distribution for different instructions in networking applications. We have simulated several applications from the NetBench suite [20]. The simulations are performed using the SimpleScalar simulator [17] and the processor configuration explained in Section 4.1. The results are presented in Figure 2. The figure gives the percentage of the data misses caused by different number of instructions. For example, in the *CRC* application, approximately 30% of the misses are caused by a single instruction and 80% of the misses are caused by five instructions with the highest number of misses. On average, 58% of the misses are caused by only five instructions and 87% of the misses are caused by 25 instructions. In addition, we have performed another set of experiments to see what type of data accessed causes the misses. The simulations reveal that 66% of the misses occur when the processor is reading packet data. The advantage of few instructions causing a majority of the misses is that if we can offload these instructions to the DFE, we can reduce the number of cache misses and L2 accesses (hence bus accesses) significantly.

In the next section, we present results from combined split cache / cache bypassing mechanism and discuss the disadvantages of such locality enhancement techniques. Section 2.1 explains the details of the LPT. In Section 3, we discuss the design of the DFE and show how the design options can be varied. Section 4 presents the

experimental results. In Section 5, we give an overview of the related work and Section 6 concludes the paper with a summary.

2. REDUCING L1 CACHE ACCESSES

There are several cache locality-enhancing mechanisms proposed in the literature [9, 13, 14, 25, 26]. The power implications of some of these mechanisms have been examined by Bahar et al. [4]. These techniques try to improve the performance by reducing the L1 cache misses. Since the L1 misses are reduced, intuitively, these techniques should also reduce the power consumption due to less L1 and L2 cache activity.

In this section, we first examine the power implications of a representative cache locality enhancing mechanism, the combined split cache / cache bypassing mechanism proposed by Gonzalez et al. [9]. We study split cache technique because our proposed mechanism uses an advanced version of this mechanism to detect the code segments to be offloaded. In this mechanism, the L1 data cache is split into two structures, one storing data with temporal locality (temporal cache) and the other storing data with spatial locality (spatial cache). The processor uses a locality prediction table to categorize the instructions into: a) accessing data with temporal locality (**temporal instructions**), b) accessing data with spatial locality (**spatial instructions**), or c) accessing data with no locality (**bypass instructions**). The detailed simulation results will be explained in Section 4. Although the technique reduces the number of execution cycles in most applications, it does not have the same impact on the overall power consumption. In most applications, the technique increases the power consumption of the data caches. The reasons for this increase are twofold: an LPT structure has to be accessed with every data access and two smaller caches (one with larger linesize) have to be accessed in parallel instead of a single cache. Nevertheless, the overall power consumption is reduced by the technique due to the significant reduction in the bus switching activity.

In the following, we first explain the LPT as presented by Gonzalez et al. [9] and then discuss the enhancements required to implement our proposed technique.

2.1 Locality Prediction Table

The LPT is a small table that is used for data accesses by the processor in conjunction with the PC of the instructions. It stores information about the past behavior of the access: last address accessed by it, the size of the access, the miss/hit history and the prediction made using other fields of the LPT. By considering the stride and the past behavior, a decision is made about whether the data accessed by the instruction should be placed into the temporal cache, spatial cache or should be uncached.

We have modified the original LPT to accommodate the information required by the DFE. We have added three more fields to the LPT: start address of the code to be offloaded to the DFE (*sadd* field), end address of the code to be offloaded (*eadd* field) and the variables (registers) required to execute the offloaded code (*lreg* field). Assuming 32-bit address and a register file of size 32, these three fields require additional 128^1 bits for each line in the LPT. The functions of these fields will be explained in the following section, where we discuss the design of the DFE.

¹ The *lreg* field has 2-bits for each of the register as explained in Section 3.1. We have assumed 32 registers in the execution cores, hence 64 bits are required by the *lreg* field.

3. DATA FILTERING ENGINE

Figure 4 presents the DFE design. The DFE is an execution core with additional features to control the passing of the memory data to the bus. If the memory request has originated from an execution core, the pass gate is opened and the request transfers to the bus as usual. If the request has originated from the DFE, the controller closes the pass gate and forwards the data to either DFE data cache or DFE instruction cache (in the experiments explained in Section 4, the DFE is equipped with 2 KB data and instruction caches, compared to the 4 KB of data and instruction caches in the execution cores). There are two more differences between the general-purpose execution cores and the DFE. First, the DFE controller is equipped with additional logic to check whether the code executed requests a register value not generated by DFE or communicated to it from the execution cores. In such a case, the DFE communicates to the master to create an interrupt for the execution core that offloaded the code segment to the DFE. Second, the DFE has a code-management unit (CMU) to keep track of the origin of the code executed.

```
sum = 0;
for (i = 0; i < 100; i++)
    sum += array[i];
```

Figure 3. A code segment showing the effectiveness of DFE

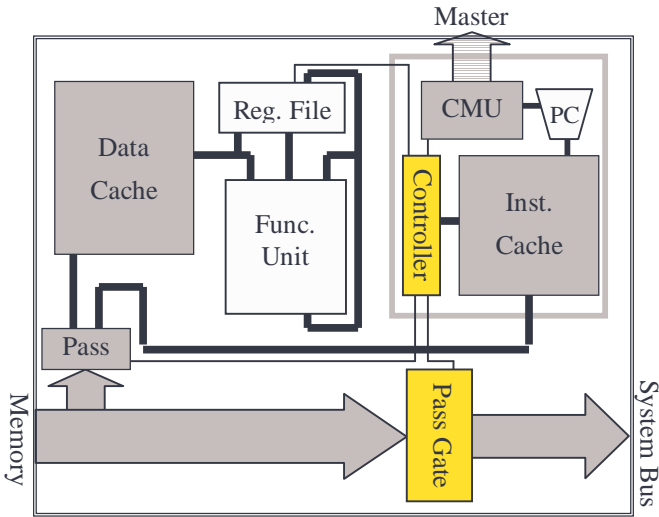


Figure 4. DFE Design.

The DFE is located between the on-chip L2 cache and the execution cores. It is activated/deactivated by the execution cores. When active, it executes a code segment that is determined by the core and communicates the necessary results back to the core. Consider the code segment in Figure 3. If the code segment is executed in one of cores, the core has to read the complete array, which means that the system bus has to be accessed several times. If the array structure is not going to be used again, these accesses will result in unnecessary power consumption in the processor. If this code segment is executed on the DFE, on the other hand, the bus will be accessed only to initiate the execution and to get the result (`sum`) from DFE. Besides reducing the number of bus accesses, offloading this segment can also have positive impact on the execution cores cache because the replacements due to accessing `array` will be prevented. Note that the code segments to be executed on the DFE are not limited to loops. We discuss which code segments should be offloaded to the DFE in Section 3.1.

3.1 Determining the DFE code

When an execution core detects an instruction that accesses low-temporal data (instructions categorized as spatial or bypass instructions by the LPT), it starts gathering information about the code segment containing the instruction. In the first run, the loop that contains the candidate PC is detected. After executing the load, the execution core checks for branch instructions that jump over the examined PC (the destination of this branch is the start of the containing loop) or procedure call/return instructions. The PC of this instruction is stored in the *eadd* field of the LPT and corresponds to the last instruction in the code to be offloaded. If the core found a containing loop, then the destination of the jump is stored in the *sadd* field, which corresponds to the start of the code to be offloaded. If there is no containing loop (the core detected a procedure call/return instruction) then the examined PC is stored in the *sadd* field. Once the start and end addresses are detected, the execution core gathers information about the register values required by the code to be offloaded. The source registers for each instruction are marked as *required* in the LPT if they are not generated within the code. Destination register is marked as *generated* if it was not marked *required*. This marking is performed with the help of the *lreg* field in the LPT². Once a code segment is completely processed, it is stored in the LPT. If the execution reaches the start address again, the code is directly offloaded to the DFE. To achieve this, we use a comparator that stores the five last offloaded code segments. If the PC becomes equal to one of these entries, the corresponding code is automatically offloaded. Note that the process of looking for code to offload can be turned off after a segment is executed several times. For the example code segment in Figure 3, the method first determines the load accesses to the `array` as candidates. Then, the loop containing the load (in this case this is the whole for loop) is captured by observing the branches back to the start of the loop. Finally, the whole loop is migrated to the DFE in the third iteration of the loop.

There are limitations to which code can be offloaded to the DFE. We do not offload segments that contain store instructions to avoid a cache coherency problem. In addition, the offloaded code must be contained within a single procedure (this is achieved by checking procedure call/return instructions). There are also some limitations related to the performance of the offloading. We offload a segment only if the number of *required* registers is below a certain *register_limit*. When a code segment is offloaded to the DFE, it is going to access the L2 cache for the code segment, so it might not be beneficial to offload large code segments. Therefore, we do not offload code segments that are larger than a *codesize_limit*. By changing these parameters, we modify the aggressiveness of the technique.

3.2 Executing the DFE code

When the execution core decides to offload a code segment, it sends the start and end addresses of the segment to the DFE along with the values of the required registers. This communication is achieved through extension to the ISA with the instructions that initiate the communication between the cores and the DFE. Therefore, no additional ports are required on the register file. After receiving a request, DFE first accesses the L2 cache to retrieve the instructions and then starts the execution. If during the execution of the segment, the DFE uses a register neither generated by the segment nor

² The *lreg* field is 2-bits and represent the states: *required*, *generated*, *required and generated*, and *invalid* (or not used).

communicated to the DFE, it generates an interrupt to the core that offloaded the segment to access the necessary register values. Such an exception is possible because the register values required by the DFE is determined by the previous executions of the code segment. Hence, with different input data, it is possible to execute a segment

that was not executed during the determination of the DFE code. When the execution ends (execution tries to go to below the *eadd* or above the *sadd*), the DFE communicates the necessary register values to the core (i.e. the values generated by the DFE).

Table 1. NetBench applications and their properties: *arguments* are the execution arguments, *# inst* is the number of instructions executed, *# cycle* is the number of cycles required, *# il1 (dl1) acc* is the number of accesses to the level 1 instruction (data) cache, *# l2 acc* is the level 2 cache accesses.

Application	Arguments	# inst. [M]	# cycle [M]	# IL1 acc [M]	# DL1 acc [M]	# L2 acc [M]
crc	crc 10000	145.8	262.0	219.0	59.8	0.6
dh	dh 5 64	778.3	1663.1	1009.1	364.7	38.4
drr	drr 128 10000	12.9	33.5	22.8	7.9	1.1
drr-l	drr 1024 10000	34.7	80.2	60.1	23.3	5.0
ipchains	ipchains 10 10000	61.7	160.2	103.9	26.2	3.6
md5	md5 10000	209.1	474.7	296.8	73.2	11.0
nat	nat 128 10000	11.4	26.7	17.3	5.6	1.2
nat-l	nat 1024 10000	33.2	74.2	55.0	21.1	5.1
rou	route 128 10000	14.2	32.0	23.3	7.1	0.9
rou-l	route 1024 10000	36.8	81.7	62.6	22.8	5.0
snort-l	snort -r defcon -n 10000 -dev -l ./log -b	343.0	925.6	515.0	132.2	33.4
snort-n	snort -r defcon -n 10000 -v -l ./log -c sn.cnf	545.9	1654.1	893.7	219.7	56.2
ssl-w	openssl NetBench weak 10000	329.0	832.1	441.1	152.0	31.8
tl	tl 128 10000	6.9	15.7	11.8	3.9	0.7
tl-l	tl 1024 10000	30.3	67.1	52.2	19.9	4.7
url	url small_inputs 10000	497.0	956.7	768.9	249.1	10.0
average		193.1	458.7	284.5	86.8	13.0

Table 2. DFE code information: number of different dfe code segments (dfe codes), average number of instructions in segments (size), average number of register values transferred for each segment (reg req), total number of register values transferred to/from the DFE (trans), fraction of instructions executed in the dfe (ratio),

App.	Dfe codes	size	reg req	trans [K]	ratio [%]
crc	2	31.2	1.5	73.2	0.03
dh	27	61.2	2.1	116.2	0.06
drr	96	28.9	2.2	32.2	5.88
drr-l	96	28.9	2.2	182.5	5.71
ipchains	113	77.8	1.8	12.3	1.86
md5	82	22.1	1.7	19.5	2.94
nat-l	67	20.1	2.3	180.1	5.77
nat	67	20.1	2.3	28.9	6.20
rou-l	39	54.2	2.2	184.8	5.72
rou	39	54.2	2.2	34.6	6.11
snort-l	146	19.4	3.4	53.1	4.35
snort-n	161	16.5	3.1	383.3	9.62
ssl-w	95	32.7	1.6	90.5	1.66
tl-l	24	43.1	2.2	179.9	5.89
tl	24	43.1	2.2	28.0	7.03
url	130	19.6	1.8	283.8	5.13
average	75	35.8	2.2	117.7	4.62

The execution core also sends a *task_id* when the code is activated. *Task_id* consists of *core_id*, which is a unique identification number of the execution core and *segment_id*, which is the id of the process offloading the code segment. *Task_id* uniquely determines the process that has offloaded the segment. When the task is complete, DFE uses the *task_id* (stored in CMU) to send the results back to the execution core. For the cores, the process is similar to accessing data from L2 cache. There might be context switches in the execution core after it sends a request, but the results will be propagated to the correct process eventually. After execution of each segment, the DFE data cache is flushed to prevent possible coherence problems.

4. EXPERIMENTS

We have performed several experiments to measure the effectiveness of our technique. In all the simulations, the SimpleScalar/ARM [17] simulator is used. The necessary modifications to the simulator are implemented to measure the effects of the LPT and the DFE. Due to the limitations of the simulation environment the execution core becomes idle when it offloads a code segment. In an actual implementation, the core might start executing other processes, which will increase the system utilization. We simulate the applications in the NetBench suite [20]. Important characteristics of the applications are explained in Table 1.

We report the reduction in the execution cycles, bus accesses and the power consumption of the caches, and overall power consumption. *The DFE is able to reduce the power consumption due to the smaller number of bus accesses and the reduction in the cache accesses.* Each will be discussed in the following.

4.1 Simulation Parameters

We first report the results for a single-core processor. The base processor is similar to StrongARM 110 with 4 KB, direct-mapped L1 data and instruction caches with 32-byte linesize, and a 128 KB, 4-way set-associative unified L2 cache with a 128-byte linesize. The LPT in both split cache experiments and our technique is a 2-way, 64 entry cache (0.5K). For split cache technique, the temporal cache is 4K with 32-byte lines and the spatial cache is 2K with 128-byte lines. In our technique, the execution core has a single cache equal to the temporal cache (4K with 32-byte lines) and DFE data cache is the similar to the spatial cache (2K with 64-byte lines). The DFE also has a level 1 instruction cache of size 2KB (32-byte lines). The latency for all L1 caches is set to 1 cycle, and all the L2 cache latencies are set to 10 cycles. The DFE is activated using L2 calls in the simulator; hence, the delay is 10 cycles to start the DFE execution.

We calculate the power consumption of the caches using the simulation numbers and cache power consumption values obtained from the CACTI tool [27]. We modified the SimpleScalar to gather information about the switching activity in the bus. The bus power consumption is calculated by using this information and the model developed by Zhang and Irwin [28]. To find the total power consumption of the processor, we use the power consumption of each section of the StrongARM presented by Montanaro et al. [21]. The power consumptions of the caches are modified to represent the caches in our simulations. The overall power consumption is the sum of the power of the execution cores and the shared resources. For the DFE experiments, the register limit is set to four and the maximum offloadable code size is 200 instructions.

4.2 Single-Core Results

For the first set of experiments, we compare the performance of four systems to the base processor explained in Section 4.1:

- System 1)** same processor with a 4 KB, 2-way set associative level 1 data cache (**2-way**),
- System 2)** same processor with a direct-mapped 8 KB level 1 data cache (**8KB**),
- System 3)** a processor with split cache (**LPT**), and
- System 4)** a processor using the proposed DFE method (**DFE**).

Table 2 gives the number of different code segments offloaded, the size of each segment, the number of registers required by each call, the total number of register values transferred to/from the DFE to execute the offloaded code and the fraction of instructions executed by the DFE. Note that the code size is the number of instructions between the first instruction in the segment and the last instruction in

the segment and does not correspond to the number of instructions executed.

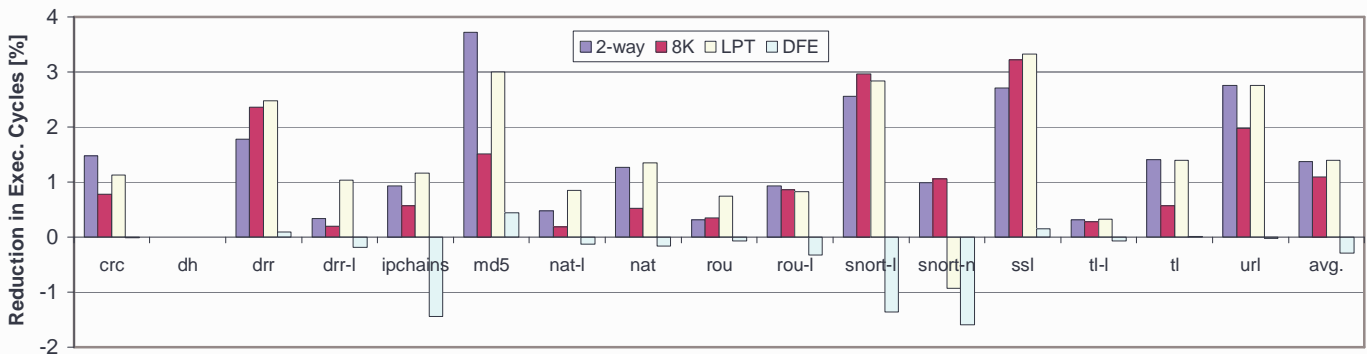
Figure 5 summarizes the results for execution cycles. It presents the relative performance of all systems with respect to the base processor. The proposed method increases the execution cycles by 0.29% on average. None of the systems have a positive impact on the performance of the *DH* application, because there are only a few data cache misses in this application (the L1 data cache miss rate is almost 0%), hence it is not effected by any of these techniques. We see that the DFE approach can increase the execution cycles by up to 1.6%. This is mainly due to the imperfect decision making about which code segments to offload. Specifically, the DFE might be activated for a code segment that contains loads to data that is used again (i.e. data with temporal locality). In such a case, the execution cycles will be improved because the data will be accessed by the cores and the DFE generating redundant accesses. If the data processed by a DFE segment is used again, then the performance might be degraded.

Figure 6 presents the effects of the techniques in the total power consumption of the data caches. For the proposed technique this corresponds to the power consumption of the level 1 data cache and the LPT structures on the execution core, level 1 data cache of the DFE and the level 2 cache. Our proposed technique is able to reduce power consumption by 0.98% on average.

The reduction in the number of bit switches on the bus accesses are presented in Figure 7. We see that the proposed mechanism can reduce the switching activity by as much as 46.8% and by 26.5% on average. The reduction in bus activity for the LPT mechanism is 18.6% on average. Similarly, the 8 KB and 2-way level 1 caches reduce the bus activity by 9.46% and 12.1%, respectively.

The energy-delay product [10] is presented in Figure 8. The DFE technique improves the energy-delay product by as much as 15.5% and by 8.3% on average. The split cache mechanism, on the other hand, has a 5.4% lower energy-delay product than the base processor.

We see that in almost every category the split cache technique performs better than a system with 8 KB cache or the 2-way associative 4 KB cache because the networking applications exhibit a mixture of accesses with spatial and temporal locality. Hence using a split cache achieves significant improvement in the cache performance. This class of applications is also amenable to our proposed structure, because spatial accesses are efficiently utilized by the DFE.



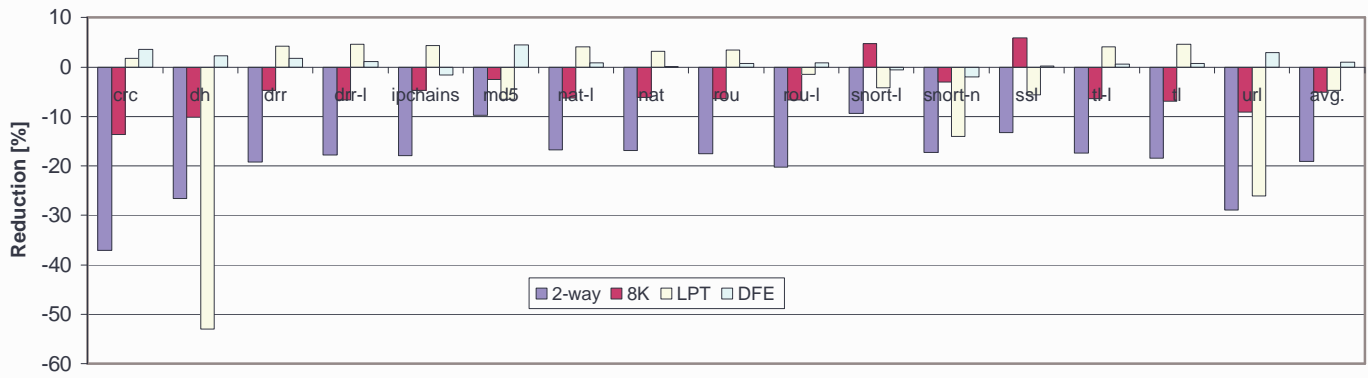


Figure 6. Reduction in power consumption of data caches (sum of level 1, level 2 and DFE data caches when applicable)

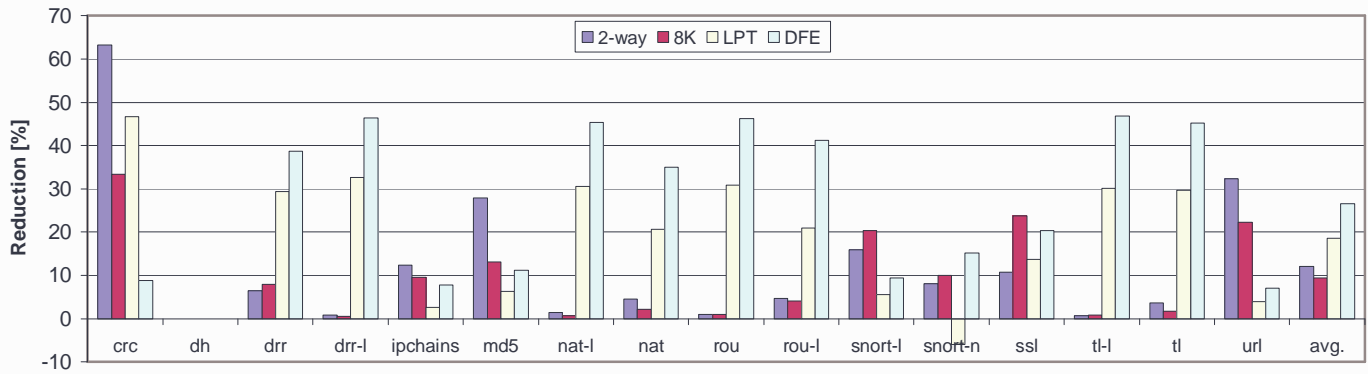


Figure 7. Reduction in number of bit switches on the system bus

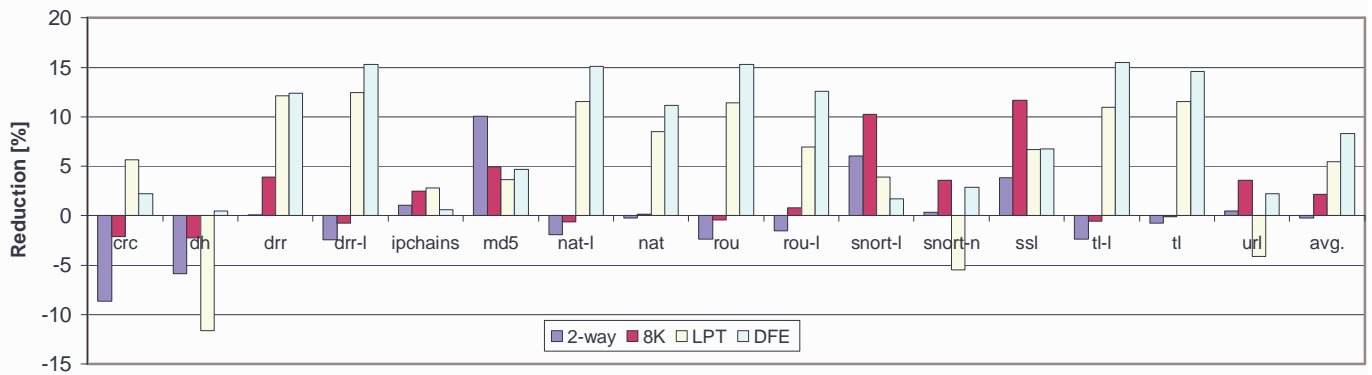


Figure 8. Reduction in energy-delay product

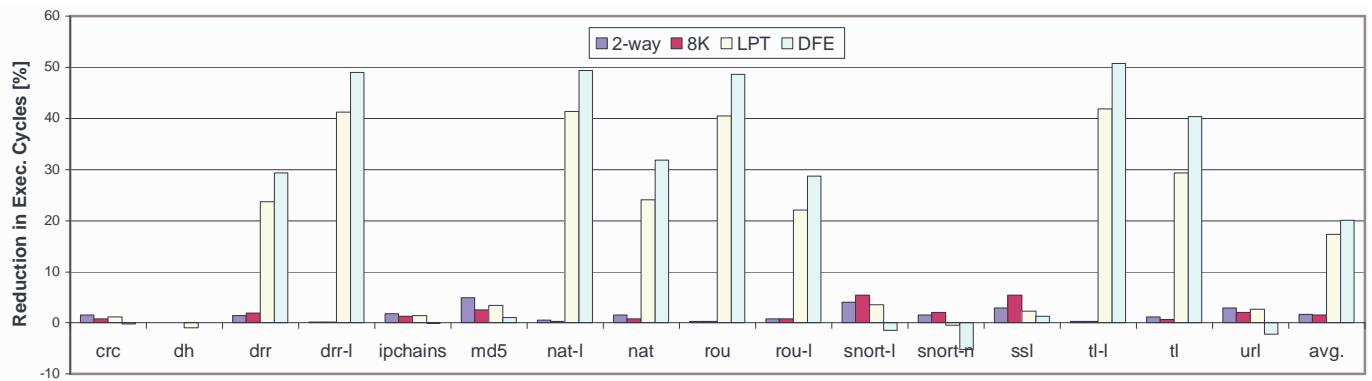


Figure 9. Reduction in execution cycles for a processor with 4 execution cores.

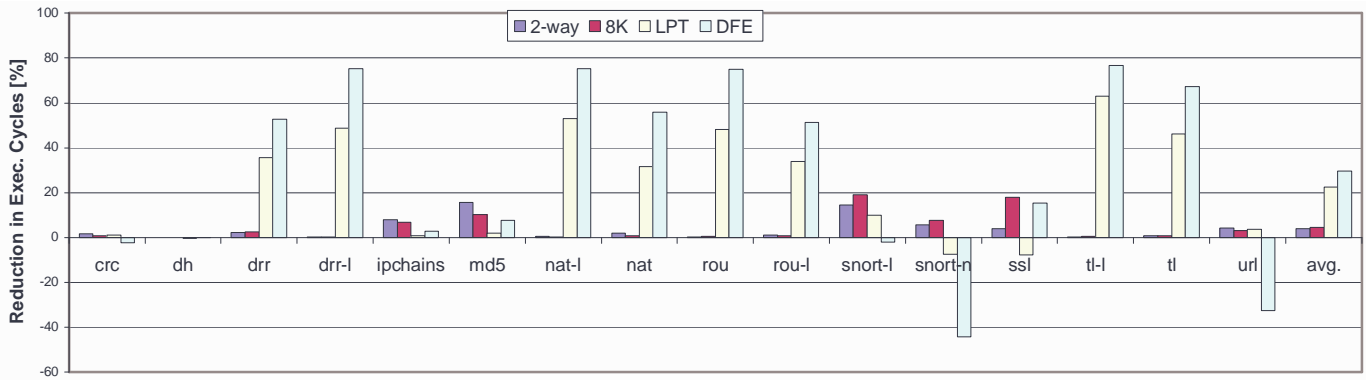


Figure 10. Reduction in execution cycles for a processor with 16 execution cores.

4.3 Multiple-Core Results

Two important issues need to be considered regarding use of the DFE in a multi-core design. First, since the DFE is a shared resource, it might become a performance bottleneck if several cores are contending for it. On the other hand, if the DFE reduces the bus accesses significantly (which is another shared resource), it might improve performance due to less contention for the bus. Note that the system bus is one of the most important bottlenecks in chip multiprocessors. Although all the techniques have small effect on the performance of a single-core processors, the performance improvement can be much more dramatic for multi-core systems due to the reduction in the bus accesses as shown in Figure 7.

To see the effects of multiple execution cores, we have first designed a trace-driven multi-core simulator (MCS). The simulator processes event traces generated by the SimpleScalar/ARM. In this framework, SimpleScalar is used to generate the events for a single-core, which are then processed by the MCS to account for the effects of global events (bus and the DFE accesses). The MCS finds the new execution time using the number of cores employed in the system and the events in the traces. Each execution core runs a single application, hence there is no communication between execution cores. Although this does not represent the workload in some multi-core systems, where a single application has control over all the execution cores, this is a realistic workload for most of the NPUs. In our experiments, we simulate the same application in each execution core. Each application has a random startup time and they are not aware of the other cores in the system, hence the simulations realistically measure the activity in shared resources. We report results for processors with 4 execution cores and 16 execution cores. Note that the cores do not execute any code and wait for the results if they performed a DFE call.

The improvement in the execution cycles for a processor with 4 execution cores are presented in Figure 9. The proposed DFE mechanism reduces the number of execution cycles by as much as 50.8% and 20.1% on average. The split cache technique is reducing the execution cycles by 17.2% on average.

Figure 10 presents the results for a processor with 16 execution cores. The DFE is able to improve the performance by as much as 76.7% and 29.7% on average. The technique increases the execution cycles for snort and url applications. For the url application, the execution cores utilize the DFE (5.13%), but are not able to reduce the bus activity. Therefore, as the number of

execution cores is increased, the cores have to stall for bus and DFE contention, resulting in an increase in the execution cycles. For snort application, on the other hand, the DFE usage is very high (9.62%). Hence, although the bus contention is reduced, the contention for DFE increases the execution cycles. Note that we can overcome the contention problem by using an adaptive offloading mechanism: if the DFE is occupied, the execution core continues with the execution itself, hence the code is offloaded only if the DFE is idle. If DFE becomes overloaded, the only overhead in such an approach would be the checking of the DFE state by the execution cores. Another solution to the contention problem might be to use multiple DFEs. However, such techniques are out of the scope of this paper.

We also measured the energy consumption for multiple cores. However, increasing the number of execution cores did not have any significant effect on the overall power consumption, because each additional execution core increases the bus, the DFE, and the level 2 cache activity linearly, hence the ratios for different system did not change significantly.

5. RELATED WORK

Streaming data, i.e. data accessed with a known, fixed displacements between successive elements has been studied in the literature. McKee et al. [19] propose using a special stream buffer unit (SBU) to store the stream accesses and a special scheduling unit for reordering accesses to stream data. Benitez and Davidson [6] present a compiler framework to detect streaming data. Our proposed architecture does not require any compiler support for its tasks. In addition, the displacement of accesses in some networking applications is not fixed due to the unknown nature of the packet distribution in the memory.

Several techniques have been proposed to reduce the power consumption of high-performance processors [2, 5, 15, 24]. Some of these techniques use small energy-efficient structures to capture a portion of the working set thereby filtering the accesses to larger structures. Others concentrate on restructuring the caches [2]. In the context of multiple processor systems, Moshovos et al. [22] propose a filtering technique for snoop accesses in the SMP servers.

There is a plethora of techniques to improve cache locality [9, 13, 14, 25, 26]. These techniques reduce L1 data cache misses by either intelligently placing the data into it or using external structures. In our study, however, we change the location of the computation of the low-temporal data accesses. Most of these

techniques could be used to detect the low temporal data for our proposed method.

Active or smart memories have been extensively studied [3, 8, 16]. However, such techniques concentrate on off-chip active memory in contrast to out methods, which improve the performance of on-chip caches. Therefore the fine-grain offloading is not feasible for such systems.

6. CONCLUSION

Network Processors are powerful computing engines that usually combine several execution cores and consume significant amount of power. Hence, they are prone to performance limitations due to excessive power dissipation. We have proposed a technique for reducing power in multi-core Network Processors. Our technique reduces the power consumption in the bus and the caches, which are the most power consuming entities in the high-performance processors. We have shown that in networking applications most of the L1 data cache misses are caused by only a few instructions, which motivates the specific technique we have proposed. Our technique uses a locality prediction table to detect load accesses with low temporal locality and a novel data filtering engine that processes the code segment surrounding the low temporal accesses. We have presented simulation numbers showing the effectiveness of our technique. Specifically, our technique is able reduce the overall power consumption of the processor by 8.6% for a single-core processor. It is able to reduce the energy-delay product by 8.3% and 35.8% on average for a single-core processor and for a processor with 16 cores, respectively.

We are currently investigating compiler techniques to determine the code segments to be offloaded to the DFE. Static techniques have the advantage of determining exact communication requirements between the code remaining in the execution cores and the segments offloaded to the DFE. However, determining locality dynamically has advantages. Therefore, we believe that better optimizations will be possible by utilizing the static and dynamic techniques in an integrated approach.

References

1. Abraham, S. G., R. A. Sugumar, D. Windheiser, B. R. Rau, and R. Gupta. Predictability of Load/Store Instruction Latencies. In Proceedings of *Twenty-sixth International Symposium on Microarchitecture (MICRO-26)*, Dec. 1993.
2. Albonesi, D. H. Selective cache ways. In Proceedings of *Int. Symposium of Microarchitecture*, Nov. 1999, Haifa / Israel.
3. Asthana, A., M. Cravatts, and P. Krzyzanowski. Towards a Programming Environment for a Computer with Intelligent Memory. In Proceedings of *Proc. of the Parallel Architectures and Compilation Techniques*, Aug. 1994, Montreal / Canada.
4. Bahar, R. I., G. Albera, and S. Manne. Power and Performance Tradeoffs using Various Caching Strategies. In Proceedings of *International Symposium on Low Power Electronics and Design*, Aug. 1998, Monterey / CA.
5. Bellas, N., I. Hajj, C. Polychronopoulos, and G. Stamoulis. Architectural and compiler support for energy reduction in the memory hierarchy of high performance processors. In Proceedings of *Intl. Symposium on Low Power Electronics and Design*, Aug. 1998.
6. Benitez, M. E. and J. W. Davidson. Code Generation for Streaming: An Access/Execute Mechanism. In Proceedings of *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 1991, Los Alamitos / CA.
7. C-port Corp. C-5 Network Processor Architecture Guide. C5NPD0-AG/D, May 2001.
8. D. Patterson, et al., *A Case for Intelligent RAM: IRAM*, in *IEEE Micro*. April 1997.
9. Gonzalez, A., C. Aliagas, and M. Valero. A Data Cache with Multiple Caching Strategies Tuned to Different Types of Locality. In Proceedings of *International Conference on Supercomputing*, 338-347, July 1995.
10. Gonzalez, R. and M. Horowitz, *Energy dissipation in general purpose microprocessors*. IEEE Journal of Solid-State Circuits, 1996. **31**(9): p. 1277-84.
11. Halfhill, T. R., *Intel Network Processor Targets Routers*, in *Microprocessor Report*. Sep. 13, 1999.
12. IBM Corp. IBM PowerNP NP4GS3 Network Processor Datasheet. IBM technical library, np3_DLTOC.fm.08, May 2001.
13. Johnson, T. L. and W. W. Hwu. Run-Time Adaptive Cache Hierarchy Management via Reference Analysis. In Proceedings of *24th International Symposium on Computer Architecture (ISCA)*, 315-326, June 1997, Denver, CO.
14. Jouppi, N. P. Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache Prefetch Buffers. In Proceedings of *25 Years {ISCA}: Retrospectives and Reprints*, 388--397, 1998.
15. Kin, J., M. Gupta, and W. H. Mangione-Smith. The Filter Cache: an energy efficient memory structure. In Proceedings of *Intl. Symposium on Microarchitecture*, Dec. 1997, Research Triangle Park / NC.
16. Kogge, P. M., T. Sunaga, E. Retter, et al. Combined DRAM and Logic Chip for Massively Parallel Applications. In Proceedings of *16th IEEE Conference on Advanced Research in VLSI*, March 1995, Raleigh / NC.
17. SimpleScalar LLC. SimpleScalar Home Page. <http://www.simplescalar.com>
18. Mangione-Smith, W. H. and G. Memik, *Network Processing: Applications, Architectures and Examples*. Tutorial at *International Symposium on Microarchitecture*, Austin / TX. Dec. 2001.
19. McKee, S. A., R. H. Klenke, K. L. Wright, W. A. Wulf, M. H. Salinas, J. H. Aylor, and A.P. Batson, *Smarter Memory: Improving Bandwidth for Streamed References*, in *IEEE Computer*. July 1998. p. 54-63.
20. Memik, G., W. H. Mangione-Smith, and W. Hu. NetBench: A Benchmarking Suite for Network Processors. In

Proceedings of *International Conference on Computer-Aided Design (ICCAD)*, pp. 39-42, Nov. 2001, San Jose / CA.

21. Montanaro, J., *et al.*, A 160-MHz, 32-b, 0.5-W CMOS RISC microprocessor. *IEEE Journal of Solid-State Circuits*, 1996. **31**(11): p. 1703-14.
22. Moshovos, A., G. Memik, B. Falsafi, and A. Choudhary. JETTY: Snoop filtering for reduced power in SMP servers. In Proceedings of *International Symposium on High Performance Computer Architecture (HPCA-7)*, Jan 2001, Toulouse / France.
23. Panda, P. R. and N. D. Dutt. Reducing Address Bus Transitions for Low Power Memory Mapping. In Proceedings of *EDTC-96: IEEE European Design and Test Conference*, pp. 63-67, March 1996, Paris / France.
24. Powell, M. D., S. H. Yang, B. Falsafi, K. Roy, and T. N. Vijaykumar. Gated-Vdd: A circuit technique to reduce leakage in cache memories. In Proceedings of *Intl. Symposium on Low Power Electronics and Design*, July 2000.
25. Rivers, J. A. and E. S. Davidson. Reducing conflicts in direct-mapped caches with a temporality-based design. In Proceedings of *International Conference on Parallel Processing*, 154-63 vol.1, 1996.
26. Tyson, G., M. Farrens, J. Matthews, and A. R. Pleszkun, *Managing Data Caches Using Selective Cache Line Replacement*. *International Journal of Parallel Programming*, 1997. **25**(3): p. 213--242.
27. Wilton, S. and N. Jouppi. An enhanced access and cycle time model for on-chip caches. July 1995.
28. Zhang, Y. and M .J. Irwin. Energy-Delay Analysis for On-Chip Interconnect at System Level. In Proceedings of *IEEE Computer Society Workshop on VLSI*, 1999.