Application Domains for Fixed-Length Block Structured Architectures

Lieven Eeckhout[†] Tom Vander Aa[‡] Bart Goeman[†] Hans Vandierendonck[†]
Rudy Lauwereins[‡] Koen De Bosschere[†]
[†]ELIS, Ghent University, Belgium
[‡]ESAT, KULeuven, Belgium

Abstract

In order to tackle the growing complexity and interconnects problem in modern microprocessor architectures, computer architects have come up with new architectural paradigms. A fixed-length block structured architecture (BSA) is one of these paradigms. The basic idea of a BSA is to generate blocks of instructions, called BSA-blocks, statically (by the compiler) and executing these blocks on a decentralized microarchitecture. In this paper, we focus on possible application domains for this architectural paradigm. To investigate this issue, we have set up several experiments with 43 benchmarks coming from the SPECint95, the SPECfp95, the MediaBench suite, plus a set of MPEG-4 like algorithms. The main conclusion of this paper is twofold. First, multimedia applications are less control-intensive than SPECint95 benchmarks and more control-intensive than SPECfp95 benchmarks. As a result, a compiler for a BSA will find more opportunities to fill BSA-blocks with instructions from the actually executed control flow paths for SPECfp95 than for multimedia applications; and more for multimedia applications than for SPECint95. Second, 16 instructions per BSA-block is appropriate for all application domains. Larger BSA-blocks on the other hand, result in higher branch misprediction rates for most applications and lead to a less effective use of the virtual window size.

1. Introduction

Scaling out-of-order architectures to higher levels of parallelism is possible by increasing the dimensions of the various structures in the architecture. However, this rapidly becomes infeasible due to the increasing complexity and the ever growing impact of interconnects on performance under technology scaling. Therefore, computer architects have come up with new architectural paradigms that are based on decentralization or partitioning. The basic concept of decentralization is quite simple: instead of providing one big and thus slow processor core, partitioned microarchitectures are built up with various small and thus very fast engines that communicate through long and thus relatively slow interconnects. The problem computer engineers then have to solve, is to find a partitioning that makes an optimal tradeoff between fast engines and slow interconnects. Several decentralized paradigms have been proposed in the literature, such as multiscalar architectures [1], trace processors [2], the clustered Alpha 21264 [3] and fixed-length block structured architectures [4]. The basic idea of the fixed-length block structured architectural paradigm is to generate blocks containing a maximum number of instructions statically (by the compiler) and to execute these blocks as atomic units on a decentralized microarchitecture. In previous work [4, 5], we have focused on and we have extensively discussed the advantages and the feasibility of fixed-length block structured architectures: its decentralized microarchitecture, its easier fetching mechanism, its possibility to eliminate hard-to-predict branches through the use of predication and its reduced register file pressure.

In this paper, we will focus on the application domains for fixed-length block structured architectures. What types of applications will get most profit from running on a fixed-length block structured processor architecture? And how should the microarchitectural parameters be fine-tuned per application domain in order to obtain optimal performance? To answer these questions we have set up several experiments with 43 benchmarks coming from the control-intensive integer SPECint95 suite, the compute-intensive floating-point SPECfp95 suite, the multimedia and signal processing MediaBench suite, plus a set of MPEG-4 like algorithms. The main conclusion of this paper is twofold. First, multimedia applications are less control-intensive than SPECint95 benchmarks, but more control-intensive than SPECfp95 benchmarks. This is due to their larger basic block sizes compared to SPECint95 benchmarks (under comparable branch misprediction rates), and their smaller basic block sizes and their higher branch mis-

0-7695-0954-1/01 $10.00 © 2001 IEEE

prediction rates compared to SPECfp95. As a result, a compiler will find more opportunities to form BSA-blocks containing useful instructions (from the actually executed control flow path) for SPECfp95 benchmarks, than for multimedia applications; and more for multimedia applications than for SPECint95 benchmarks. Second, blocks containing more than 32 instructions are inappropriate due to their reduced branch predictability and their less effective use of the virtual window size, which is reflected in reduced performance in some cases and marginal performance increase (compared to smaller blocks, e.g. 16-instruction blocks) in most cases.

This paper is organized as follows. Section 2 presents the fixed-length block structured architectural paradigm and briefly discusses its advantages and its disadvantages. The methodology used in this paper, namely statistical modeling, as well as the benchmarks are discussed in section 3. The core issues of this paper are discussed and evaluated in section 4. Finally, we conclude in section 5.

2. Block Structured Architecture

A block structured architecture (BSA) is an architectural paradigm that was first presented in [6] by Melvin and Patt. A particular form of BSAs, namely *fixed-length* BSAs[1], have been further refined by Neefs [7]. In a BSA, instructions are statically grouped into fixed-length *blocks*. The number of instructions included in a block is bounded by a maximum, e.g. 16 instructions, and the instructions can be taken from various basic blocks. Since no control flow is allowed within a block, predication [8] is needed to transform intra-block control flow into data flow. Once the instructions to be included in a block are determined, register renaming is performed by the compiler to obtain a static single assignment form, which maximizes the attainable parallelism within a block. Besides an *instructions* section, a BSA block also contains an *in* section, an *out* section and a *branches* section. The in section itemizes the inter-block registers that are used by the instructions in the block. The out section specifies the inter-block registers that are defined and live on exit; i.e. the out section maps intra-block registers to inter-block registers. The in and out sections take care of the inter-block data communication, while the branches section specifies the inter-block control flow.

To illustrate the principles of a block, an example is shown in Figure 1. The registers denoted as `r` and `i` are inter-block and intra-block registers, respectively. Inter-block registers are architecturally visible, whereas intra-block registers are only visible within a block. In the instructions section, when `i1` equals zero, predicate register `p1` is set; otherwise it is cleared. If `p1` (also called the

[1]For the remainder of this paper, we only consider fixed-length BSA, which will be denoted as BSA.

```
                      | IN:     r1, r2, r3
c = c + 1;            | INSTR: add r3,1 -> i1 (==0,p1)
if ( c == 0 ) {       |        (p1)  add r1,1 -> i2
   a = a + 1;         |        (p1)  add r2,1 -> i3
   b = b + 1;         |        (~p1) add r1,2 -> i4
}                     | OUT:   i1 -> r3
else {                |        (p1)  i2 -> r1
   a = a + 2;         |        (~p1) i4 -> r1
}                     |        (p1)  i3 -> r2
                      | BRNCH: fall-through
```

Figure 1. Example: source code (on the left) and BSA code (on the right).

guard) is true, one is added to `r1` and `r2`, otherwise 2 is added to `r1`. In the out section, when `p1` is true, `i2` is mapped to `r1`, and `i3` to `r2`; otherwise, `i4` is mapped to `r1`. In all cases, `i1` is mapped to `r3`.

2.1. Microarchitecture

When executing only one block at a time, performance will be low. Therefore, we have chosen for a microarchitecture that allows for parallel execution of multiple blocks, leading us to a particular implementation of the control-dependence based decentralized paradigm. This means that several units of work of a sequential program are (speculatively) executed in parallel on separate processing elements. The atomic unit of work in a BSA is called a *block*, and a processing element is called a *block engine*, see Figure 2. In a BSA, a head and a tail pointer indicate the block engine that executes the earliest and the latest assigned block, respectively. At any time, there is only one block engine executing a block non-speculatively, which is indicated by the head pointer. The branch predictor predicts the follower block of the one currently being executed in the block engine indicated by the tail pointer. The predicted block is then fetched and assigned by a sequencer to the follower block engine indicated by the tail pointer. This head-and-tail mechanism is also used in multiscalar architectures [1] and trace processors [2].

As suggested by the use of intra-block and inter-block registers in a BSA, a distinction is made between intra-block and inter-block communication. Inter-block communication is concerned with the propagation of data values between different block engines. Between adjacent block engines, data values are propagated through associative logic; between non-adjacent block engines, values are propagated through the register file, see Figure 2. Intra-block communication, on the other hand, is related to the communication flow within a single block engine, and follows the data flow execution policy, which is made possible thanks to the static single assignment form. An instruction that resides in the instruction window, is selected to be ex-

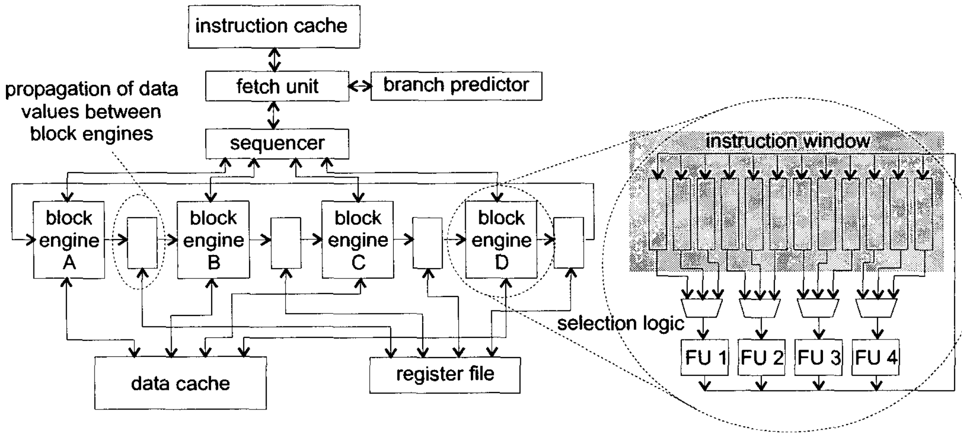

Figure 2. Possible microarchitecture of a BSA.

ecuted on a functional unit when all its operands are available (data-flow). An important feature concerning the intra-block communication is the speculative execution of predicated instructions. This means that predicated instructions can be executed before the value of their guard is known[2], eliminating the true data dependency introduced by predication [8]. Correct program semantics are guaranteed by mapping the correct data values in the out section, see Figure 1.

2.2. Advantages

A block structured architecture has several advantages over traditional superscalar processors [4, 5]:

- **Decentralization.** First of all, a BSA is an answer to the scalability problem of traditional out-of-order architectures. In future chip technologies wiring delay will become a major obstacle in boosting clock frequencies in traditional superscalar architectures due to large window sizes and wide issue widths [9]. The central idea is to have small (and thus very fast) block engines interconnected by relatively long (and thus slow) interconnections.

- **Easier fetching.** Since the length of a BSA block is fixed, fetching instructions will be easier. We do not need to predict multiple branches in a single clock cycle. Moreover, we do not need to fetch from different parts in the instruction cache within a single cycle. Easier fetching was the major motivation for Melvin and Patt [6] to come up with a block structured ISA.

[2]i.e. instructions are only selected for execution when the guard is true or is still unknown.

- **Predication.** Predication was proven to be an interesting technique to eliminate unbiased branches and to expose multiple execution paths to the processor [8]. Indeed, predicated instructions from multiple paths are speculatively executed regardless of the guard's value; the correct register values are then committed if the corresponding guards are true in the out section.

- **Fewer register file ports.** Franklin and Sohi [10] showed that the lifetime of register instances is restricted; i.e. the temporal distance measured in the number of instructions between the use of a register instance and its creation is restricted. Thus, grouping nearby instructions into blocks, will keep most inter-operation communication within block boundaries. And since in a BSA only inter-block communication passes to the register file, see Figure 2, the register file will need fewer access ports, resulting in a smaller register file access time.

2.3. Disadvantages

A block structured architecture also has some disadvantages:

- **Lower IPC for a given virtual window size.** Since instructions are committed per block—instead of individually as is the case in an out-of-order architecture—the virtual window will be less efficiently utilized, resulting in lower IPC for a given virtual window size. The smaller the maximum number of instructions in a block, the smaller the IPC (number of instructions retired per cycle) degradation will be.

- **Slower inter-block communication.** The inter-block communication in a multi-block BSA will be slower—measured in number of cycles—than the intra-window communication in traditional architectures. This will decrease IPC, especially for small blocks due to a larger amount of inter-block communication.

- **Higher memory bandwidths and larger instruction cache required.** Predication, register renaming by the compiler, and the inclusion of various sections in a block require more encoding bits per instruction. As a consequence, instruction caches of larger sizes with higher access times are required as well as higher memory bandwidths. One possible solution to overcome this disadvantage is into organizing the instruction cache in several cache banks. Each cache bank contains a part of a BSA block. Fetching a block then involves accessing the various cache banks (with the same address) to fetch the various parts of the block from the appropriate cache banks and coalescing them to form the BSA block wanted. Another possibility would be to use compressed BSA-blocks.

- **Compilation is hard.** To obtain good performance results, blocks should be filled with useful instructions as much as possible. This is certainly a challenge for the compiler. A preliminary study of the compiler requirements is described in [11].

3. Methodology

Section 3.1 details the technique that will be used in the evaluation section of this paper, namely statistical modeling. Section 3.2 discusses the 43 benchmarks that will be used; section 3.3 itemizes the microarchitectural assumptions being made in the experiments.

3.1. Statistical modeling

To determine IPC in a reliable way, detailed simulations are required on a cycle-accurate functional simulator executing optimal BSA code. There are two problems with this technique: first, a highly optimizing compiler as well as a detailed simulator need to be developed which are time-consuming tasks. Second, once we have set up this experimental environment, a huge amount of simulations need to be done that are time consuming as well, since several hundreds of millions or even billions of instructions need to be simulated for various processor configurations for various workloads. Therefore, we decided to perform an early design stage evaluation using a novel technique, namely statistical simulation [12, 13, 14, 15], which allows fast simulation (a steady state solution is reached after simulating only a few million instructions) and quite accurate IPC estimates.

Basically, statistical simulation [16] consists of three phases, see Figure 3. First, programs are analyzed to extract

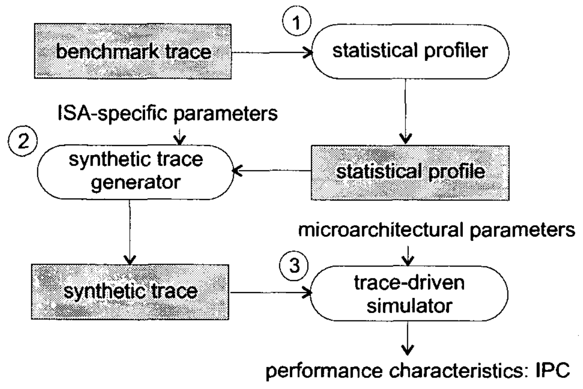

Figure 3. Statistical modeling and simulation. The various ISA-specific and microarchitectural parameters will be varied in the experiments of section 4.

a statistical profile. In a second phase, a synthetic instruction trace is produced à la Monte Carlo using that statistical profile. Subsequently, this instruction trace is fed into a trace-driven simulator modeling the architecture, which yields performance characteristics, such as IPC (number of useful instructions executed per clock cycle).

During statistical profiling (phase 1), several program execution statistics are extracted from a benchmark trace: the instruction mix, the distribution of the dependency distance between instructions (i.e. the distance counted in the number of instructions in the trace between the producer and the consumer of a register instance), the probability that a memory operation is dependent on the x-th memory operation before it in the trace through a memory data value, the distribution of the basic block size, the probability that a branch instruction is biased to the same branch outcome.

A synthetic BSA-trace is generated in phase 2. This is done in two steps. In the first step, the synthetic BSA-block construction step, basic blocks are added to the BSA-block until the maximum BSA-blocksize is reached; basic blocks are added to the most likely control flow path. The synthetic BSA-block formation is illustrated in Figure 4 through an example. First basic block `a` (path probability $p = 1.0$) is included in the BSA-block, then `b` ($p = 0.65$), then `e` ($p = 0.40$), then `c` ($p = 0.35$), and finally `d` ($p = 0.25$). Notice that an actual compiler would add predicates to the instructions of the basic blocks to guarantee correct program semantics, i.e. `p1` to `b`, `~p1` to `c`, `p1&p2` to `d`, and `p1&~p2` to `e`. The path probabilities used by the synthetic BSA-block generator were derived from profile information. Once a synthetic BSA-block has been created, the actually executed path is pointed à la Monte Carlo using the path probabilities that were calculated in the first step.

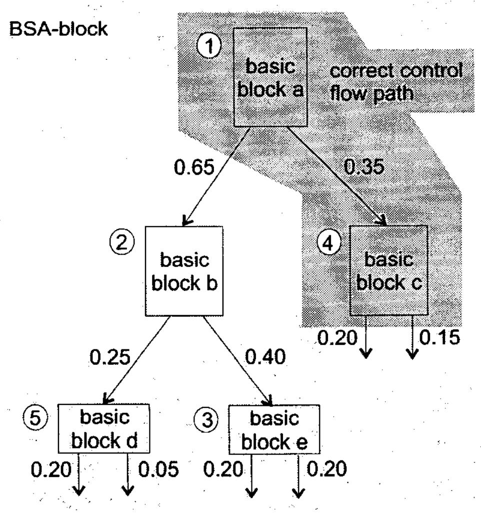

Figure 4. Building up a synthetic BSA-block as a collection of basic blocks. The probabilities determine the probability that the arrow is part of the actually executed path. The numbers in the circles show the order in which the corresponding basic blocks are included. The basic blocks shown in dark gray are marked to be part of the correct control flow path.

For example, in Figure 4, basic blocks 1 and 4 were pointed to be part of the correct control flow path. As a result, the instructions from basic blocks 1 and 4 are the only *useful* instructions in the BSA-block.

The last phase in the statistical modeling methodology is trace-driven simulation: synthetic BSA-traces are simulated on a trace-driven simulator which yields performance results, namely IPC.

3.2. Benchmarks

The benchmarks used to collect statistical profiles are listed in Table 1. A total number of 43 benchmarks were used in our analysis, coming from the SPECint95 suite (control-intensive integer applications), the SPECfp95 suite (compute-intensive floating-point applications)[3], the signal and multimedia processing MediaBench suite [19], plus a set of five algorithms from different MPEG-4 subdomains: an advanced audio decoder aac_dec (the audio standard in MPEG-4), a BIFS (Binary Interchange Format for Scenes in MPEG-4) parser t2b and generator b2t, a video decoder h263 aimed at real-time video conferencing, a simple image rendering-like program cav_det and a highly optimized 3-D image reconstruction algorithm snake [17, 18]. All these benchmarks were instrumented with ATOM [20], a binary instrumentation tool, on a DEC 500au station with an Alpha 21164 processor. The Alpha architecture is a load/store architecture with 32 integer and 32 floating-point registers, each of which is 64 bits wide. All the benchmarks were built with the DEC C compiler version 6.1 with optimization level -O4 and linked statically with the -non_shared flag.

[3]see `http://www.spec.org`

benchmrk	source	description	dyn. cnt
li	SPECint95	Xlisp interpreter	1,000M
go	SPECint95	go-playing game	593M
compress	SPECint95	text compressing	1,000M
gcc	SPECint95	GNU C compiler 2.5.3	1,103M
m88ksim	SPECint95	M88100 simulator	1,000M
ijpeg	SPECint95	image (de)compression	1,000M
perl	SPECint95	Perl interpreter	2,945M
vortex	SPECint95	object oriented database	1,000M
applu	SPECfp95	partial differential equations	261M
apsi	SPECfp95	wheather prediction	1,446M
fpppp	SPECfp95	quantum chemistry	237M
hydro2d	SPECfp95	Navier Stokes equations	4,514M
mgrid	SPECfp95	3-D potential field	9,259M
su2cor	SPECfp95	Monte-Carlo method	1,000M
swim	SPECfp95	Shallow water equations	432M
tomcatv	SPECfp95	Vectorized mesh generation	1,000M
turb3d	SPECfp95	Turbulence modeling	1,000M
wave5	SPECfp95	Maxwell's equations	1,000M
adpcm_c	MediaBench	speech compression	14M
adpcm_d	MediaBench	speech decompression	11M
g721_e	MediaBench	voice compression	641M
g721_d	MediaBench	voice decompression	590M
epic	MediaBench	image compression	235M
unepic	MediaBench	image decompression	17M
gsm_e	MediaBench	GSM encoding	432M
gsm_d	MediaBench	GSM decoding	131M
mpeg2_e	MediaBench	MPEG-2 video encoding	1,553M
mpeg2_d	MediaBench	MPEG-2 video decoding	190M
gs	MediaBench	ghostscript	126M
pgp_e	MediaBench	cryptography encoding	130M
pgp_d	MediaBench	cryptography decoding	110M
mesa	MediaBench	3-D graphics library	128M
cjpeg	MediaBench	image compression	110M
djpeg	MediaBench	image decompression	41M
rasta	MediaBench	speech recognition	25M
aac_dec		Advanced Audio Codec	164M
b2t		BIFS generator	23M
t2b		BIFS parser	27M
h263		real-time video decoder	267M
cav_det		image cavity detection	97M
snake	[17, 18]	3-D image reconstruction	126M

Table 1. The benchmarks used with their dynamic instruction count (in millions).

3.3. Microarchitecture

Several assumptions were made concerning the microarchitecture:

- An integer operation has an execution latency of 1 cycle, a load 3 cycles (this includes address calculation and data cache access), a multiplication 8 cycles, a FP operation 4 cycles, a single and double precision FP division 18 and 31 cycles, respectively. All operations are fully pipelined, except for the division.

- Clearing a block from a block engine and starting the execution of a new block takes three clock cycles: (i) dispatching the new block to the block engine, (ii) reading the register values from the register file and (iii) selecting instructions to be executed.

- A branch misprediction incurs a five cycle penalty: (i) calculating the new block address, (ii) fetching the correct block from the I-cache, (iii) conducting the fetched block to the appropriate block engine, (iv) reading data values from the register file and (v) selecting the instructions to be executed.

- Dynamic memory disambiguation is assumed with re-execution which means that loads and stores are issued out-of-order. However, when a memory dependency violation is detected, the violating load need to be re-executed as well as all its dependent instructions. In [5], it is shown that this strategy is required in a BSA to avoid a huge performance drop due to memory dependencies. A similar technique is implemented in the Alpha 21264 [3].

4. Evaluation

First, the various benchmarks are characterized and evaluated in section 4.1. Subsequently, in section 4.2, we will verify how well the compiler is capable of filling the BSA-blocks with instructions from the actually executed control flow paths. Finally, we will quantify how this affects performance in section 4.3.

4.1. Benchmarks

To characterize the benchmarks and to clarify the results that will be presented in the following sections, we will discuss two important characteristics, the instruction mix and the control-intensitivity.

Instruction mix The instruction mix for the various benchmarks is shown in Figure 5. From this graph we can conclude that the SPEC benchmarks are more memory-intensive (an arithmetic average of 40.6% and 37.7% load/stores for SPECint95 and SPECfp95, respectively) than the multimedia applications (29.2% load/stores). However, some multimedia applications have load/store rates that are comparable to the SPEC95 load/store rates, e.g. mpeg2_d (42.3%), snake (52%) and rasta (39.6%). The SPECint95 suite also has significantly more control instructions (14%) than the multimedia applications (8.5%) and the SPECfp95 suite (3.6%). However, some multimedia applications have scores that are comparable to the SPECint95 benchmarks, e.g. g721 (12%), epic (16%), gs (11.5%) and b2t2b (15.5%). Note also that some multimedia applications, e.g. epic, mpeg2, mesa, b2t2b and snake, contain floating-point operations.

Control-intensitivity A good measure for the control-intensitivity is the number of instructions between two mispredicted control instructions. This characteristic is an important characteristic since it gives an idea of how well a dynamically scheduled processor or an optimizing compiler will be able to build an instruction window in order to schedule independent instructions. And this characteristic is dependent on two other characteristics:

- the number of instructions between two control instructions (conditional and unconditional branches, indirect jumps, subroutine calls and returns), which we will call the dynamic basic block size. The SPECint95 benchmarks have a dynamic basic block size of 7.3 on (geometric) average. The number of instructions between control instructions is significantly larger for multimedia applications, namely 14.3. The SPECfp95 benchmarks have the highest dynamic basic block size, namely 25.

- the control misprediction rate which is the probability that a prediction based on the most likely behavior of a control instruction is incorrect [21]. The control misprediction rate R is measured as follows:

  $$R = 1 - \sum_{i=1}^{N_c} P(C_i) \cdot P_a(C_i)$$

  with N_c the static number of control instructions, $P(C_i)$ the probability that control instruction C_i is executed. $P_a(C_i)$ is defined as the prediction accuracy of control instruction C_i:

  $$P_a(C_i) = max\{P\,[\,T_j \mid C_i\,] \mid T_j \in \mathcal{T}_{C_i}\}$$

  with $\mathcal{T}_{C_i}$ the set of possible targets of control instruction C_i. In other words, $P_a(C_i)$ is the maximum probability of the targets of control instruction C_i. The SPECint95 benchmarks and the multimedia applications have the same control misprediction rate of 9.1% (geometric average). The control misprediction rate for the SPECfp95 is significantly lower, namely 6%.

These two characteristics can now be combined to one single characteristic, the average number of instructions between two mispredicted control instructions, by dividing

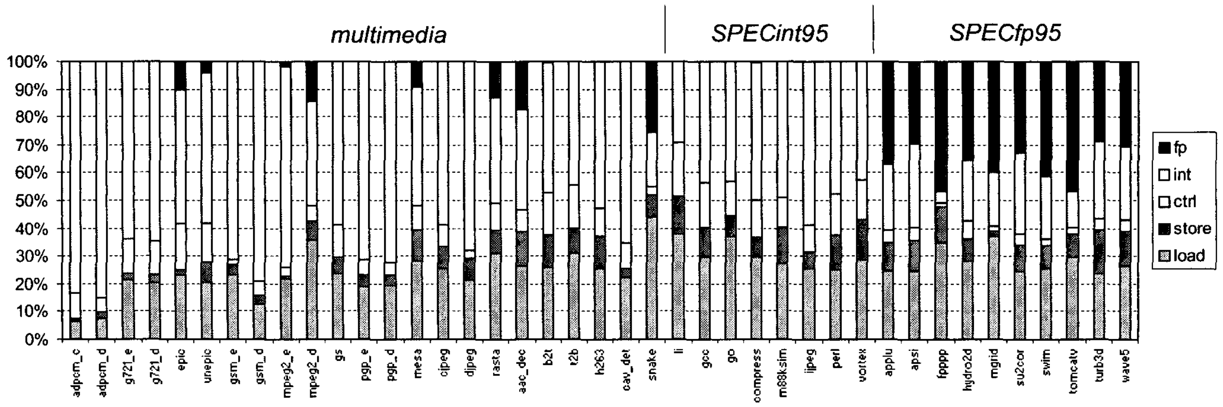

Figure 5. The instruction mix.

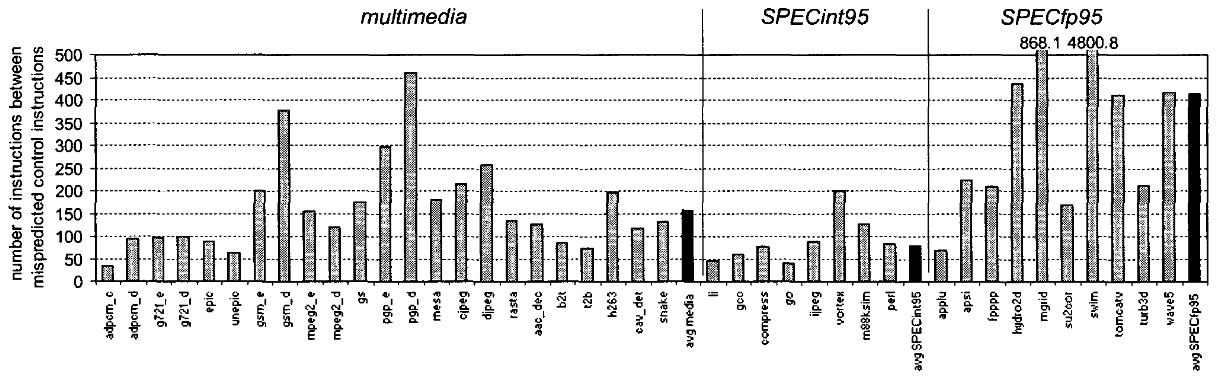

Figure 6. The number of instructions between two mispredicted branches.

the dynamic basic block size with the control misprediction rate. This characteristic is presented in Figure 6 for the various benchmarks. The (geometric) average number of instructions between two mispredicted control instructions is 80.1, 156.9 and 415.3 for the SPECint95 suite, the multimedia applications and the SPECfp95 suite, respectively. From this graph we can conclude that SPECfp95 and SPECint95 have the greatest and the lowest opportunity to build large instruction windows, respectively.

Several interesting observations can be made from Figure 6: the number of instructions between two mispredicted control instructions is quite small (35.3 instructions) for adpcm_c since its control misprediction rate 30.6% is extremely high (the highest score over all the benchmarks), with a near average dynamic basic block size of 10.8 instructions. pgp_d has a high score in Figure 6 due to its low misprediction rate 6.2% and its large 22.2 dynamic basic block size. For the SPECint95 suite, go and vortex have the lowest and highest score due to their misprediction rates 20.7% and 3.4%, respectively; however, their dynamic basic block size is near the SPECint95 average, 8.0 and 6.9, respectively. For the SPECfp95 suite, applu has an extremely low score compared to the average SPECfp95 score due to its high misprediction rate 24.3% and its small dynamic basic block size 16.6; mgrid and swim on the other hand have high scores in Figure 6 due to their large dynamic basic block size, 56.8 and 45.2, and their misprediction rates, 6.5% (near the SPECfp95 average) and 0.9%, respectively.

4.2. BSA-block formation

An interesting aspect to measure is how many useful instructions will be included in a BSA-block, i.e. what fraction of the instructions in a BSA-block belong to the actually executed control flow path. In other words, we want to measure how well the compiler is capable of filling BSA-blocks with useful instructions. Obviously, this will be application-dependent and will be determined by the basic block size and the branch predictability. The fraction of useful instructions in a BSA-block is shown in Figure 7 as a function of the BSA-block size. Applications with larger dynamic basic block sizes and lower control mispre-

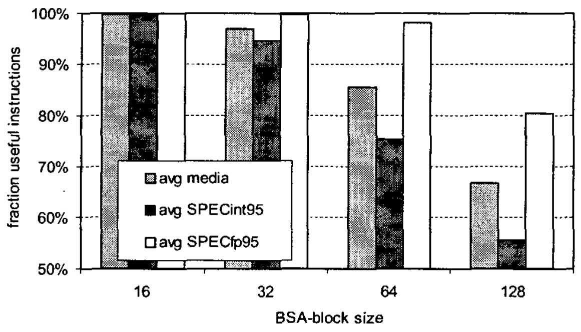

Figure 7. The fraction of instructions that belong to the actually executed control flow path.

diction rates will have a larger fraction of useful instructions than applications with smaller dynamic basic block sizes or lower control misprediction rates. Consequently, SPECfp95 applications have larger fractions of useful instructions in a BSA-block than multimedia applications and SPECint95 benchmarks; multimedia applications have larger fractions of useful instructions per BSA-block than SPECint95 benchmarks.

Another important aspect is the predictability of the multi-way branch at the end of a BSA-block. The predictability of a BSA-block is defined as the maximum probability of the exit edges. For example, in Figure 4, there are six exit edges of which the maximum probability is 20%. As a consequence, the predictability of that BSA-block is defined to be 20%. We have measured the predictability of the BSA-blocks as a function of the BSA-block size; the results are shown in Figure 8. The predictability for 16-instruction blocks is quite high for most benchmarks. For 32-instruction blocks on the other hand, the predictability is significantly lower for several SPECint95 benchmarks (li, gcc, compress and go) and some multimedia applications (adpcm_c, b2t and t2b). The predictability for 64-instruction blocks is low for nearly all benchmarks, except for most SPECfp95 benchmarks.

4.3. Performance

How are these characteristics reflected in performance? To quantify this, we have done several simulations for various microarchitectures by varying the BSA-block size (16 and 32 instructions) and the number of block engines[4]. The BSA-block size affects performance in two ways. First, the branch predictability is lower for larger BSA-blocks. Second, larger BSA-blocks make less effective use of the virtual window size, see section 2.3. The branch prediction accuracy of the statistical branch predictor, which predicts BSA-block exit edges, can be imposed in our simulator and was set to the predictability of a BSA-block (see previous section) for each particular benchmark. The results of these experiments are presented in Figure 9. The upper row of Figure 9 shows some typical examples, namely gs, t2b and g721_d. Due to the high predictability of the 32-instruction blocks for gs (92%), the performance for 32-instruction blocks is higher than for the 16-instruction blocks with a branch predictability of 94%. For t2b on the contrary, performance is significantly lower for 32-instruction blocks than for 16-instruction blocks due to the low predictability of 32-instruction blocks (61% compared to 91% for 16-instruction blocks) and the lower fraction of useful instructions per block, see Figure 7. Another typical case that we have encountered is shown in the graph for g721_d. Performance for 32-instruction blocks increases slower as a function of the number of block engines than for 16-instruction blocks; as a result, performance is lower for 32-instruction blocks than for 16-instruction blocks for the configuration with 16 block engines. The bottom row of Figure 9 presents the (geometric) average IPC obtained for the three application domains investigated, i.e. multimedia, SPECint95 and SPECfp95. For the multimedia applications, performance is a little higher on average for 32-instruction blocks than for 16-instruction blocks. This is due to the average branch predictability for 32-instruction blocks (87%) which is quite high compared to the 90% branch predictability for 16-instruction blocks. However, the performance increase is marginal compared to the increased hardware resources needed (32-instruction windows with 4 functional units instead of 16-instruction windows with 2 functional units). For the SPECint95 benchmarks, performance is lower for the 32-instruction blocks due to the low branch predictability for 5 of the 8 benchmarks; the average branch predictability for 32-instruction blocks 81% is quite low compared to 91% for 16-instruction blocks. For the SPECfp95 benchmarks, 32-instruction blocks are inappropriate as well (taking the additional hardware cost into account).

5. Conclusions

In order to tackle the growing complexity and interconnection problem in modern microprocessor architectures, computer architects have come up with decentralized architectural paradigms. In previous work, we have shown the feasibility of fixed-length block structured architectures in this respect. In this paper, we have focused on the application domains for this architectural paradigm. We have done this by setting up several experiments and measurements on 43 benchmarks coming from the control-intensive integer SPECint95 suite, the compute-intensive

[4]2 and 4 functional units are provided per block engine for the 16- and the 32-instruction blocks, respectively.

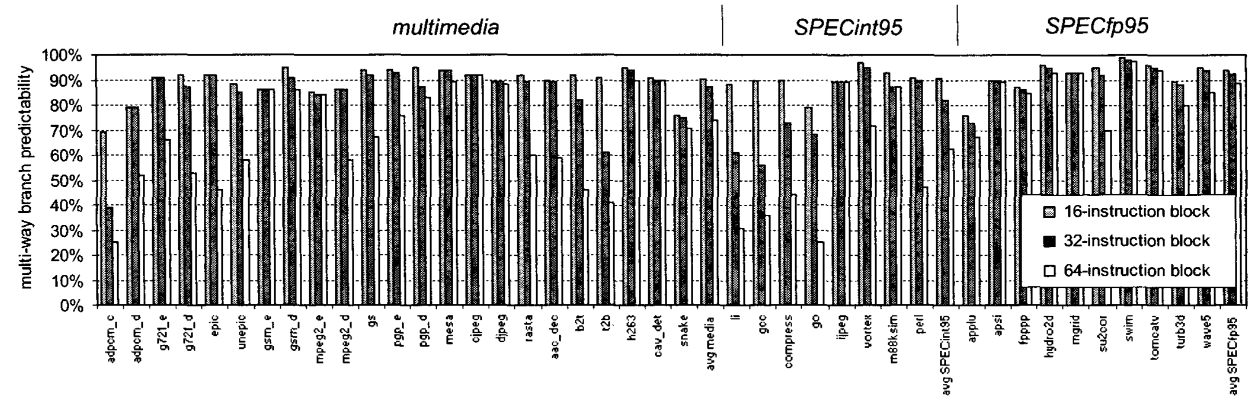

Figure 8. Multi-way branch predictability as a function of BSA-block size.

floating-point SPECfp95 suite, the multimedia and signal processing MediaBench suite, plus a set of MPEG-4 like algorithms. The main conclusion of these experiments is twofold. First, multimedia applications are less control-intensive than SPECint95 benchmarks, but more control-intensive than compute-intensive floating-point applications. This is due to their larger dynamic basic block sizes compared to SPECint95 benchmarks (under comparable control misprediction rates), and their smaller dynamic basic block sizes and their higher control misprediction rates compared to SPECfp95. This is reflected in the fraction of useful instructions that can be included in a BSA-block. Second, the fraction of useful instructions per BSA-block is quite high for BSA-blocks containing up to 32 instructions for any application domain (more than 95% for most benchmarks). The branch predictability for 32-instruction blocks on the other hand, is significantly lower for several applications (especially for SPECint95) than the branch predictability for 16-instruction blocks. Thus, in spite of the high fraction of useful instructions per block, 32-instruction blocks are inappropriate due to their lower branch predictability and due to the less effective use of the virtual window size.

6. Acknowledgments

Lieven Eeckhout and Hans Vandierendonck are supported by a grant from the Flemish Institute for the Promotion of the Scientific-Technological Research in the Industry (IWT). Tom Vander Aa is supported by the FWO-poject G.0036.99 and the TI Elite project. Bart Goeman is supported by the FWO-project G.0036.99 on block structured architectures for multimedia signal processing. Koen De Bosschere is research associate with the Fund for Scientific Research-Flanders.

References

[1] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar. Multiscalar processors. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 414–425, June 1995.

[2] E. Rotenberg, Q. Jacobson, Y. Sazeides, and J. Smith. Trace processors. In *Proceedings of the 30th Annual International Symposium on Microarchitecture*, pages 138–148, December 1997.

[3] L. Gwennap. Digital 21264 sets new standard. *Microprocessor Report*, 10(14):1–6, October 1996.

[4] L. Eeckhout, K. De Bosschere, and H. Neefs. On the feasibility of fixed-length block structured architectures. In *Proceedings of the 5th Australasian Computer Architecture Conference ACAC 2000*, pages 17–25, January 2000.

[5] L. Eeckhout, H. Neefs, and K. De Bosschere. Early design stage exploration of fixed-length block structured architectures. *Journal of Systems Architecture*, 2000. Accepted for publication.

[6] S. Melvin and Y. Patt. Enhancing instruction scheduling with a block-structured ISA. *International Journal of Parallel Programming*, 23(3):221–243, 1995.

[7] H. Neefs. A preliminary study of a fixed length block structured instruction set architecture. Technical Report Paris 96-07, Dept. of Electronics and Information Systems, University of Gent, November 1996. Available through http://www.elis.rug.ac.be/~neefs.

[8] S. A. Mahlke, R. E. Hank, J. E. McCormick, D. I. August, and W. W. Hwu. A comparison of full and partial predicated execution support for ILP processors. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 138–149, June 1995.

[9] S. Palacharla, N. P. Jouppi, and J. E. Smith. Complexity-effective superscalar processors. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 206–218, June 1997.

[10] M. Franklin and G. S. Sohi. Register traffic analysis for streamlining inter-operation communication in fine-grain parallel processors. In *Proceedings of the 22nd Annual International Symposium on Microarchitecture (MICRO-22)*, pages 236–245, December 1992.

[11] H. Neefs, K. De Bosschere, and J. Van Campenhout. Issues in compilation for fixed-length block structured instruction set architectures. In *Proceedings of the Workshop on Interaction between Compilers and Computer Architectures*, February 1997. In Conjunction with the Third International Symposium on High-Performance Computer Architecture (HPCA-3).

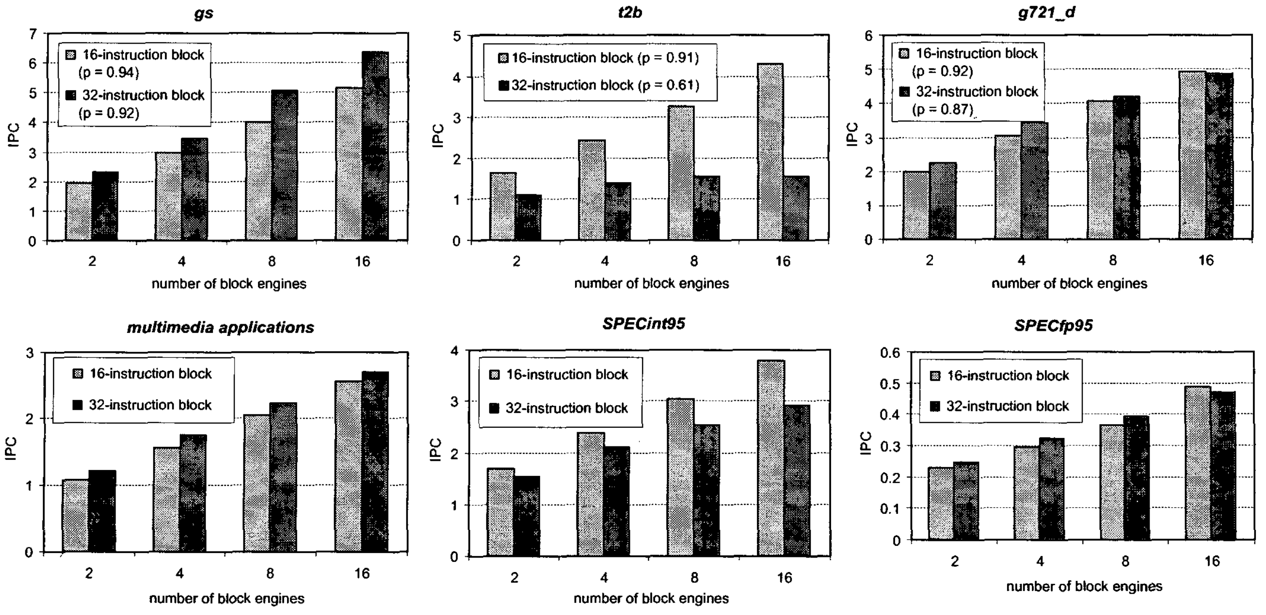

Figure 9. IPC as a function of the number of block engines and BSA-block size. Perfect caches were assumed in these experiments.

[12] D. B. Noonburg and J. P. Chen. A framework for statistical modeling of superscalar processor performance. In *Proceedings of the third International Symposium on High-Performance Computer Architecture (HPCA-3)*, pages 298–309, February 1997.

[13] R. Carl and J. E. Smith. Modeling superscalar processors via statistical simulation. In *Workshop on Performance Analysis and its Impact on Design*, June 1998.

[14] L. Eeckhout, K. De Bosschere, and H. Neefs. Performance analysis through synthetic trace generation. In *The IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS-2000)*, pages 1–6, April 2000.

[15] M. Oskin, F. T. Chong, and M. Farrens. HLS: Combining statistical and symbolic simulation to guide microprocessor design. In *Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA-27)*, June 2000.

[16] L. Eeckhout, H. Neefs, and K. De Bosschere. Estimating IPC of a block structured instruction set architecture in an early design stage. In *Parallel Computing: Fundamentals and Applications; Proceedings of the International Conference ParCo99*, pages 468–475, January 2000.

[17] M. Proesmans, L. J. Van Gool, and A. J. Oosterlinck. One shot active 3D shape reconstruction. In *Proceedings 13th ICPR International Conference on Pattern Recognition: applications & robotic systems*, volume III C, pages 336–340, August 1996.

[18] T. Van Achteren, M. Adé, R. Lauwereins, M. Proesmans, L. Van Gool, J. Bormans, and F. Catthoor. Transformations of a 3D image reconstruction algorithm for data transfer and storage optimisation. In *Proceedings of the RSP IEEE International Workshop on Rapid System Prototyping*, 1999.

[19] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. MediaBench: A tool for evaluating and synthesizing multimedia and communications systems. In *Proceedings of the 30th Annual IEEE/ACM Symposium on Microarchitecture*, pages 330–335, December 1997.

[20] A. Srivastava and A. Eustace. ATOM: A system for building customized program analysis tools. Technical Report 94/2, Western Research Lab, Compaq, March 1994.

[21] T. M. Conte and W. W. Hwu. Benchmark characterization for experimental system evaluation. In *Proceedings of the 23rd Hawaii International Conference on System Sciences*, volume 1, pages 6–18, January 1990.