Characterization of embedded applications for decoupled processor architecture

Assia Djabelkhir André Seznec
IRISA, Compus de Beaulieu
35 042 Rennes Cedex, FRANCE
adjabelk,seznec@irisa.fr

Abstract

Needs for performance on embedded applications will lead to the use of dynamic execution on embedded processors in the next few years. However, complete out-of-order superscalar cores are still expensive in terms of silicon area and power dissipation. In this paper, we study the adequation of a more limited form of dynamic execution, namely decoupled architecture, to embedded applications.

Decoupled architecture is known to work very efficiently whenever the execution does not suffer from inter-processor dependencies causing some loss of decoupling, called LOD events. In this study, we address regularity of codes in terms of the LOD events that may occur. We address three aspects of regularity: control regularity, control/memory dependency, and patterns of referencing memory data. Most of the kernels in MiBench will be amenable to efficient performance on a decoupled architecture.

Keywords: embedded processors, decoupled architecture, embedded benchmarks, MiBench, code regularity, workload characterization, loss of decoupling.

1. Introduction

Needs of embedded applications in terms of performance and programming flexibility increase in parallel with integration possibilities. Thus processors used for embedded applications (automobile, telephone, set-top boxes...) are increasingly used as both micro-controllers and DSPs. Their clock frequencies - even if they remain being set back compared to general-purpose processors - impose to resort to architectural techniques which were reserved up today to high performance processors: cache memories, instruction parallelism, very strongly pipelined execution...

However, difficulties related to conception and compilation for embedded processors slightly differ from those of general-purpose processors. While performance is the principle criterion for general-purpose processors, different criterions must also be considered for embedded processors: cost, power consumption, and performance predictability. Thus, up to now most embedded processors execute instructions in order and use no, or few, speculative and/or out-of-order execution. Using dynamic execution makes performance of embedded processors less predictable and tends to consume more power.

Even so, embedded applications are becoming more complex and their dynamic behavior may vary according to input stimuli and response time. In particular, static scheduling of control sections may limit performance. Using the complete out-of-order execution implementation on modern superscalar processors may result on a significant benefit for embedded applications. The overall objective of this study is to show that more limited dynamic execution such as the one encountered in decoupled architecture may be sufficient for many embedded applications, and therefore may be more cost effective than using the superscalar architecture.

Dynamic instruction scheduling resolves control and data dependencies at runtime. Decoupling is an optimization technique for high-performance computer architectures [4] that is a very powerful technique for minimizing the impact of memory latency, and is applicable to a wide range of applications. So, in this paper we aim to understand the behavior of embedded applications on a decoupled architecture and address aspects and constraints to design an effective embedded decoupled processor.

Decoupled architecture (Figure 1) attains high performance if its “processors” execute in a fully decoupled way, i.e. with a sufficient slip between them. Loss of decoupling (LOD) between processors constitutes the principal cause of execution penalty. The loss of decoupling occurs when inter-processor dependencies interfere with the decoupled execution. Regular applications have good behavior on decoupled architecture: decoupling is evident and no occurrence of inter-processor dependency that risks to break down the pipeline of decoupling. In this study, we analyze the characteristics of a wide spectrum of embedded applications and address regularity of codes in terms of the LOD events that may occur. We address three aspects of regu-

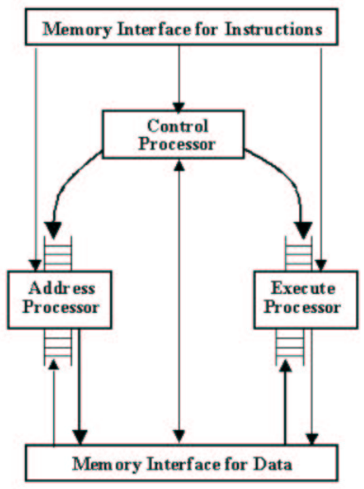


Figure 1. Access and control decoupled architecture.

larity: control regularity, control/memory dependency, and patterns of referencing memory data.

The workload evaluation is performed on MiBench embedded benchmark suites [1]. We use calvin2+DICE simulator [2] to analyze these benchmarks and extract features that will direct architectural decisions to take, in the future steps of design. We base our study on the access and control decoupled processor model proposed in [4].

The workload evaluation shows that most of kernels in MiBench may be amenable to efficient performance on a decoupled architecture. However, the study of control flow regularity shows that on few applications, mispredicted conditional branches caused by some control/memory dependencies are quite frequent, and this may result in poor performance due to loss of decoupling.

The remainder of this paper is organized as follows. In Section 2, we describe decoupled architecture model addressed by our study. Section 3 presents the various aspects of regularity we address in codes and the associated metrics we use. Section 4 provides the evaluations and discusses the results we obtain. Section 5 summarizes the conclusions and presents directions for our future work.

2. Decoupled Architecture

Decoupling is an optimization technique for high-performance computer architectures. It is a very powerful technique for minimizing the impact of memory latency, and is applicable to a wide range of programs. For instance, media applications as being very structured and regular and lend themselves well to the decoupling concept [8].

ZS-1 [6], PIPE [9], and WM [7] constitute the first decoupled access/execute architectures, which are fine-grain processors seeking to maximize performance by dividing a given program into two separate instruction streams, the Access stream and the Execute stream, and executing them on two independent cooperating processes, the memory access process and the computation (or execute) process. This allows exploiting parallelism between the two streams. The access stream consist of those instructions involved in generating memory accesses, so the moving of data to and from memory. The execute stream consists of those instructions that perform some operations on that data.

In these architectures, the addresses for memory references are generated in advance of the execution of data-related instructions. This means that memory read operations can be initiated many cycles before the read data is required for execution, and the latency of main memory read operations (or cache operations) can be hidden. This means that there is a sufficient slip between the two processors, and that execute processor will not stall waiting for memory data. Consequently, if the processor running the access stream is able to initiate, sufficiently in advance, load operations, then it can get ahead of the execute stream and the penalty due to long memory latency will be reduced or eliminated.

Bird et al. have introduced, in [4], the control decoupling concept – a further technique for increasing performance, to maximize the use of main memory bandwidth in an implementation, by exploring the control-flow graph of a program ahead of the time at which computation is required. This enables requests for packets of computation to be queued ahead of the time at which they are required, so that when one computation had finished, another is ready to take its place on the *relevant* processor.

An access and control decoupled architecture consists of three, independent and cooperating, processors communicating via queues (Figure 1): Control processor, Address processor, and Execute processor. The control processor (CP) resolves, in advance, branches and delivers the good flow of instructions to be executed. CP splits this flow into access and computation instruction streams to be processed by the appropriate processors. The address processor (AP) performs the data fetch ahead of demand to alleviate delays due to memory latency. The AP also performs indexing and other addressing operations. The execute processor (EP) operates on the data and produce results.

In this architecture, the CP runs ahead of the AP and EP, and the AP runs ahead of the EP. It is conceivable for the CP and the EP to be separated in time by several thousands of program statements. When the system is fully decoupled, one would like that the AP be typically ahead of the EP by a time equal to the largest memory latency experienced by any load operation since the last recoupling of the AP and EP.

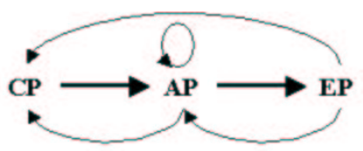


Figure 2. **Loss of decoupling events.**

Loss of decoupling (LOD)

There are a number of specific events that will cause an interruption in the pipeline flow. These occur whenever information travel against the normal direction of flow for the decoupling pipeline (the backward arrows in figure 2). At such points in the program, some degree of decoupling is lost and we therefore refer to as Loss Of Decoupling (LOD) points. In a decoupled architecture, LOD events are the principal cause of execution penalties [4]. When the system is fully decoupled, the entire physical address space appears to be accessible within one cycle; at a LOD point however a large penalty is paid.

Typically, no LOD would occur in regular programs. Consequently, our analysis of embedded application behavior will address LOD events that may occur. In Section 3, we present the main cases of inter-processor dependencies that can interfere with decoupling.

3. Workload Characterization

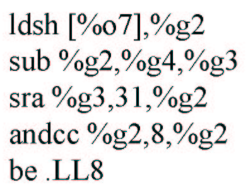
This workload characterization aims to understand behavior of embedded applications on an access and control decoupled architecture. Typically, a regular behavior means that the three processors execute in a fully decoupled way, i.e. with a sufficient slip to each other and no LOD event would occur. With an irregular behavior, the decoupled execution risks to break down more frequently, resulting on a considerable loss of performance.

In this analysis, we first address the regularity of control flow in the applications and quantify the slip between CP and AP/EP. Second, we study the case when CP must recouple with AP waiting for a memory data and propose solutions to deal with such CP/AP recoupling. Last, we extract the patterns in which data are used in the applications, in order to deal with memory latency.

3.1. Control regularity/predictability

Ideally, if the control processor is able to generate/predict the instruction flow without using external data from the access or execute processor, then it will be able to provide the AP with the access instruction stream sufficiently in advance, so that AP loads earlier memory data EP needs. However, irregular control in applications, that is conditional branches with poor predictability, would delays CP until condition evaluation is complete. This constitutes a LOD, i.e. CP recouples with AP/EP.

The impact of irregular control is particularly important if it occurs in innermost loops, as these constitute the most time consuming portion of code in most of applications. If innermost loops are regular, i.e. iterate over a large and known, or predictable, number of times, and perform simple computations and no irregular control, CU delivers instructions in the loop body and the number of iterations to AP and EP. Thus, CU would have large time to prepare the next outer loop iteration or another innermost loop. But, if irregular control occurs in innermost loops, control processor will have to make control at each iteration, which results on a high rate of LOD events.



```
ldsh [%o7],%g2
sub %g2,%g4,%g3
sra %g3,31,%g2
andcc %g2,8,%g2
be .LL8
```

Figure 3. **CMD example form adpcm coder assembler code.**

3.2. Control/Memory dependency (CMD)

One of the causes that may break down the pipeline of decoupled architecture is memory latency. For example, if EP requires data not yet ready, computation can stall for hundreds of cycles. In the worst case, CP needs results of those computations to resolve a conditional branch with poor predictability before dispatching more instruction streams. This means that the three units recouple, resulting on a considerable loss of performance. This would not be the case if the branch can be predicted.

To study this case, we define the CMD (Control/Memory Dependency) distance as the number of instructions in the computation chain of the irregular branch condition, starting from the last memory load in the chain. A small distance is an indicator that a severe LOD is likely to occur on this branch if data is not present in the cache, inducing large miss penalty.

In figure 3, we present an example of CMD from `adpcm` application. The computation chain from the load (*ldsh: short load*) to the conditional branch (*be: branch if equal*) is of 3 instructions. So, if the branch has poor predictability there is a CMD, and it is important in this case that data can be loaded without latency. Consequently, one would like that data, on which depends irregular control, be kept in cache.

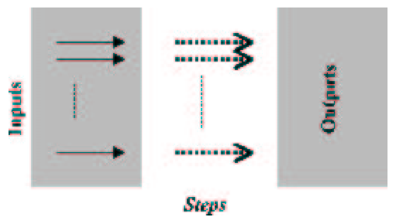


Figure 4. Flow data referencing pattern.

3.3. Data referencing pattern (DRP)

Applications use data in different patterns. We identify three categories of data whose use can cause some LOD, which are: permanent data, indirect accesses (e.g. pointers), and transitory data. Permanent data are those data EP will use permanently in computations. To save memory bandwidth, one would like to keep permanent data in cache, close to the EP.

References to memory data can depend on other memory data, which is the case with indirect memory accesses. To deal with such AP/AP dependency, one would like that pointers be kept in nearest memory locations (cache) so that address processor can initiate earlier such memory loads.

Transitory (or temporary) data are those data that remain alive only for few cycles. This is the case, for example, when a loop processes an input file, buffer by buffer. So, data are used in a flow DRP (FDRP) and the process can be represented as a direct line: input buffer - computation - output buffer (Figure 4). One would like that such transitory data do not disrupt content of cache memory. So, it may be effective to avoid loading them in the main cache. We note that transitory or permanent data can be written-read-written (WRW) within a short delay. This is the case, for example, of the buffers receiving inputs in the flow pattern: data are loaded in buffer, processed and than other data are written in that buffer. Therefore, a small auxiliary cache should be considered for these transitory data.

In summary, one would like that data causing CMD events, permanent data, and pointers be kept in cache, to limit LOD events and save memory bandwidth, while transitory data can be fetched from memory without being stored in data cache. So, one would like that transitory data be loaded in an auxiliary cache.

4. Evaluation and discussion

In this section, we present our evaluation on the MiBench embedded benchmark suite [1], which includes regularity of control, control/memory dependency, and the pattern of referencing data. To instrument codes and collect dynamic information, we use calvin2+DICE simulator [2], on a Sun Ultrasparc workstation, to which we add the necessary user-defined routines in order to collect the required characterization information.

We first present our evaluation framework: benchmarks and tools we use, then the qualitative characteristics of the MiBench applications. Provided optimizations are turned on.

These characteristics are mainly inherent to the application and its coding in high level language and mainly independent from the compiler and the ISA.

4.1. Evaluation environment

Workload evaluation we present in this paper is conducted on the MiBench embedded benchmark suite [1], defined at the University of Michigan. We use calvin2+DICE simulator [2] to extract static and dynamic code features. Programs were compiled using `gcc -O3`.

4.1.1 MiBench benchmark suites

The wide range of applications makes it difficult to characterize the embedded domain. In fact, an embedded benchmark suite should reflect this by emphasizing diversity. Embedded applications range from sensor systems on simple microcontrollers to smart cellular phones that have the functionality of a desktop machine combined with support for wireless communications.

There have been some efforts to characterize embedded workloads, most notably the suite developed by the EDN Embedded Microprocessor Benchmark Consortium (EEMBC) [3]. They have recognized the difficulty of using one suite to characterize such a diverse application domain and have instead produced a set of suites that typifies workloads in five embedded markets. Unfortunately, the EEMBC benchmarks are not readily accessible to academic researchers.

MiBench is a set of representative embedded applications for benchmarking purpose. Following the EEMBC's model, these benchmarks are divided into six suites with each suite targeting a specific area of the embedded market, which include applications from automotive and industrial control, consumer devices, networking, security, and telecommunication categories. All the programs are available as standard C source code. As inputs, most applications are provided with small and large data sets. The small data sets represent a lightweight, useful embedded application of the benchmark, while the large data set provides a more useful, real-world application. In what follows, we present an overview of MiBench benchmarks in each category.

Automotive and Industrial Control The automotive and industrial control benchmarks represent set of embedded control systems.

basicmath	simple mathematical calculation.
bitcounts	tests the bit manipulation abilities of a processor by counting the number of bits in an of integers, using five methods .
susan	is an image recognition package that recognizes corners and edges in Magnetic Resonance Images of the brain, and can smooth image.

Consumer Devices The consumer devices benchmarks represent consumer devices like scanners, digital cameras, and Personal Digital Assistants (PDAs).

jpeg	A standard, lossy image compression coder (cjpeg) and decoder (djpeg) for color and grayscale image, based on the JPEG standard;
lame	a GPL'ed MP3 encoder that supports constant, average, and variable bit-rate encoding.
tiff2bw	converts a color TIFF image to black and white image.
tiff2rgba	converts a color image in the TIFF format into an RGB color formatted TIFF image.
tiffdither	dithers a black and white TIFF bitmap to reduces the resolution and size of the image at the expense of clarity.
tiffmedian	converts an image to a reduced color palette by taking several medians of the current color palette.
typeset	a general typesetting tool, that has a front-end processor for HTML.

Office Automation The office applications are text manipulation algorithms to represent office machinery like printers, fax machines and word processors.

stringsearch	searches for given words in phrases using a case comparison algorithm.

Network Benchmarks of networking represent processors in network devices like switches and routers.

dijkstra	constructs a large graph in an adjacency matrix representation and then calculates the shortest path between every pair of nodes using repeated applications of Dijkstra's algorithm.
patricia	A Patricia trie is a data structure used in place of full trees with very sparse leaf nodes. The input data is a list of IP traffic from a highly active web server. The IP numbers are disguised.

Security The Security category includes several common algorithms for data encryption, decryption and hashing.

blowfish	is a symmetric block cipher with a variable length key.
rijndeal	was selected as the National Institute of Standards and Technologies Advenced Encryption Standard (AES). It is a block cipher with the option of 128-, 192-, and 256-bit keys and blocks.
sha	is the secure hash algorithm that produces a 160-bit message digest for a given input.

Telecommunications Many portable consumer devices are integrating wireless communications.

adpcm	Adaptive Differential Pulse Code Modulation is a simple adaptive differential pulse code modulation coder (*rawcaudio*) and decoder (*rawdaudio*).
crc32	performs a 32-bit Cyclic Redundancy Check on a file. The data input is the sound files from the *adpcm* benchmark.
fft	performs a Fast Fourier Transform and its inverse transform on an array of data.
gsm	The Global Standard for Mobile communications is the standard for voice encoding/decoding in Europe and many countries.

4.1.2 calvin2+DICE simulator

calvin2+DICE toolset for microarchitecture simulations [2] is a platform of a cost effective trace collection and on-the-fly simulation approach. The original application code is lightly annotated to provide a fast (direct) execution mode, with calvin2. An embedded instruction-set emulator, DICE, enables trace collection or on-the-fly simulations. At run time, dynamic switches are enabled from the fast mode to the emulation mode by the annotation code, and vice-versa.

calvin2 is a static code annotation tool which instruments SPARC assembly code. DICE emulates SPARC V9 instruction-set architecture (ISA) code: it manages the emulation execution mode of target programs. DICE enables simulation by calling user-defined analysis routines between each instruction emulated. Analysis routines have direct access to all information in the target program state, including complete memory state, and register values.

4.2. Regularity of control

To investigate the control flow regularity/predictability in MiBench applications, we address innermost loop characteristics, in terms of dynamic loop instruction count (DLIC) and irregular control in these loops.

We define the DLIC metric as the total number of instructions executed in the loop, which corresponds to the interval of time CP has to prepare the next outer loop iteration or another innermost loop. So, we add to the simulator the routine that detect innermost loops, at run-time, and evaluate the DLIC metric. We classify the results we obtain into four classes: DI50 represents the loops executing not more than 50 instructions, while DI+ is the class of loops executing more than 1000 instructions. Figure 5 presents the ratio of instructions in loops of each class per the number of executed instructions in the kernels. DI50 and DI200 are the classes of the loops we consider as small. DI+ is the class of loops that are large.

Predictability of branches in innermost loops is also an issue since each branch misprediction may result in a loss of

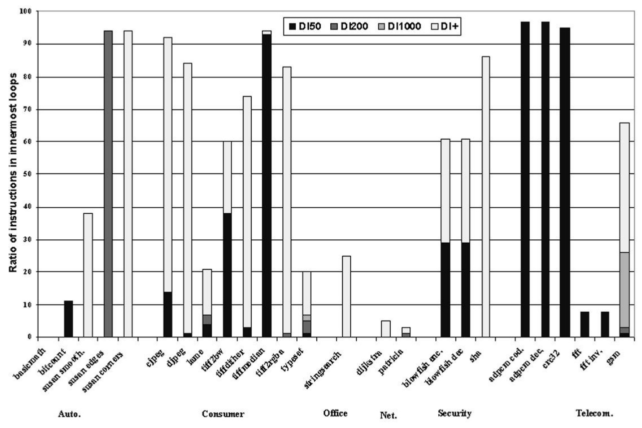


Figure 5. **Dynamic instruction count in MiBench's innermost loops.**

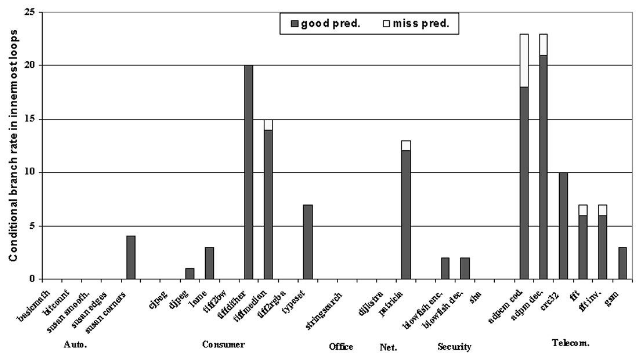


Figure 6. **Control rate in the main innermost loops of MiBench's kernels.**

decoupling. We use the 32 Kbits 2Bc-gskew hybrid branch predictor [5] integrated to the simulator, with respective history sizes of 0, 13, 9, and 11, to quantify conditional branch predictability in the main innermost loops (Figure 6).

We analyze the results below in each domain. There is no control in innermost loops of applications form the automotive domain. In `basicmath` and `bitcount`, the process is performed in functions executing simple arithmetic computations, called in the innermost loops. This does not be shown in our simulation, as we do not consider the instructions executed in called functions and treat call instructions as any other instruction in the loop. `susan` is an application of image recognition that perform lot of computations and manipulations on memory data (processed image) for which we studied three methods: `susan_smoothing`, `susan_edges`, and `susan_corners`. In `susan_smoothing`, most of the treatment is performed in the outer loops. No call occur in the main innermost loops of `susan_edges` and `susan_corners`, whose 94% of executed instructions is in their innermost loops. The main innermost loops of `susan_corners` are large and have some predictable control, which means that CP has sufficient time to go further ahead in the dynamic control flow graph of the application.

Main loops in most of benchmarks from consumer domain process inputs buffer by buffer. The process of compression/decompression of JPEG images is very regular as performed in large innermost loops that perform very few control. The main process of `lame` and `typeset` applications is executed in functions called by the innermost loops. The kernel of `tiff2bw` consists of loops with small bodies, iterating over the image width. The functions called within these loops perform some computations and no control. A control rate of up to 20% is encountered in the main innermost loops of `tiffdither` which predictability depends on some initial parameters, so predictable. `tiffmedian` seems to be irregular: innermost loops are small and some of the control encountered depends on processed inputs, and causes some mispredictions.

`stringsearch` is an algorithm that searches for a mord in a sentence. Its main innermost loop is large and performs no control but calls for a searching function that performs a set of comparisons, not always predictable as depend on the strings compared.

`dijkstra` and `patricia` execute the main processes in functions called in their innermost loops, and which perform a set of control. Some of the conditional branches encountered depend on memory data (adjacency matrix in `dijkstra` and the routine tables in `patricia`) and are quite unpredictable.

Innermost loops in encryption applications accomplish a number of logic operations and no, or few, control. The control occurring in innermost loops of `blowfish` encryption/decryption process is predictable because it depends on some parameters fixed at the initialization. `sha` is a regular application: 85% of executed instructions belongs to large innermost loops.

Innermost loops in the applications from the telecommunication domain are small and perform some control that attains a rate of 23% in the `adpcm` coder/decoder. This application seems to be the most irregular benchmark we study. The control encountered in its innermost loops is predictable at a rate of only 80% in the coder, as it depends on input stimuli.

In this evaluation, we note that some of MiBench kernels spend more time in outer loops (such as `susan_smoothing`) and called functions (such as the applications of networking). To further understand behavior of these applications, we address in the following paragraph the overall control predictability and the characteristics of mispredicted control in terms of control/memory dependency.

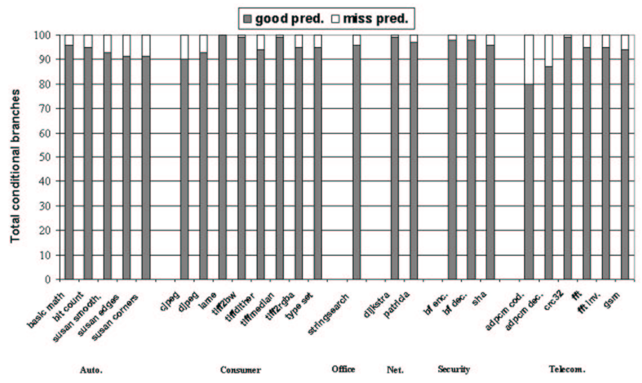


Figure 7. **Predictability in Mibench's applications.**

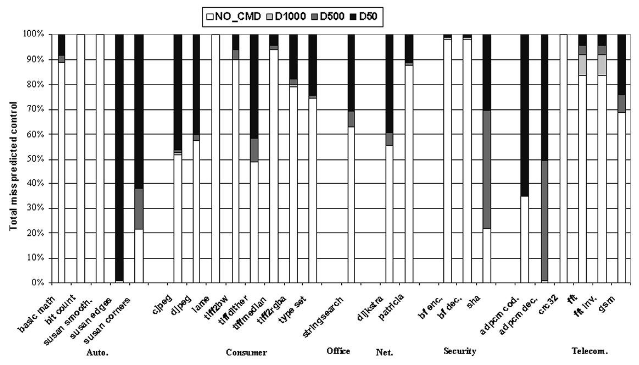


Figure 8. **CMD distance classification.**

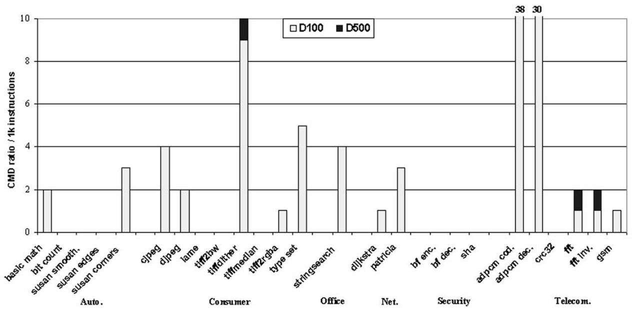


Figure 9. Ratio of CMD occurrences in MiBench's kernels.

4.3. Control/Memory dependency

In this section, we analyze the CMD distance evaluation. As defined in the paragraph 3.2, the CMD distance is the number of instructions in the computation chain of the irregular branch condition, starting from the last memory load in the chain. We address only irregular control, that is, conditional branches with poor predictability. In addition, when the distance is high than 1000 instructions, we consider that address processor has sufficient time interval to load data, in order to resolve the dependency with no, or few, penalty. The routine we define evaluates the CMD distance starting from each memory load operation until reaching the maximum value of 1000 instructions, or a conditional branch. If the reached branch is well predicted there is no dependency. The mispredicted control is classified depending on the obtained distance of control/memory dependency.

Figure 7 presents the mispredictions detected in MiBench's kernels, classified depending on the CMD distance into four classes (Figure 8): D50 represents the cases where the distance of dependency is lower than 50 instructions. D500 is the class of cases with a CMD distance ranging from 50 to 500 instructions, etc. NO_CMD represent the cases with distance larger than 1000 instructions. This classification will help us to decide of the appropriate loading policy to avoid long penalties.

Nearly all the mispredictions on `basicmath` and `bitcount` applications do not depend on memory data. Some control from `susan` depends on the image processed. This explains the rate of 10% of mispredicted control. The CMD distance evaluated at this control is lower than ten instructions in the most of cases form `susan_edges` and `susan_corners` paths.

As explained in the previous section, there are some control/memory dependencies in the process of compression/decompression of JPEG image. This is the case also of some tiff applications such as `tiffdither` and `tiff2rgba`. The control occurring in `lame` application is highly predictable because the application is very regular.

`stringsearch` performs a set of comparisons on strings, which causes some dependencies. Some irregular control occurring in the network applications, due to `patricia` tree node and `dijkstra` adjacency matrix manipulation, cause some control/memory dependencies.

We note that the main processing loop of the `adpcm` coder/decoder presents a high rate of small CMD distance. In almost of the cases, CMD distance is lower than 500 instructions. For these cases of dependency, it would be important to maintain data in nearest memory locations in order to avoid long latency of memory loads.

The CMD distance is greater in `sha` and `fft` applications. Initiating loads little in advance should be sufficient to corresponding data be ready and so avoid high misprediction penalty even if this data does not reside in cache.

Severe loss of decoupling inducing long misprediction penalty will occur when two phenomena appear: branch misprediction and short CMD distance (let us say less than 100). Figure 9 illustrates the ratio of such bad situations per kilo instructions. We note that most of kernels in MiBench do not suffer from such dependencies.

However, on some paths from jpeg, tiffdither, typeset, stringsearch, and adpcm applications, the control/memory dependencies occur more frequently, which may result in a considerable loss of decoupling. For instance, there is a pick in adpcm with a ratio of 30-38 occurrences per kilo instructions.

As conclusion, apart a few exceptions, the applications in MiBench should not suffer from very high misprediction penalty on a decoupled processor.

4.4. Data Referencing Pattern

To study the DRP on MiBench applications, we extract each category of data presented in section 3.3, which are: permanent data, transitory data, and pointers (indirect memory accesses). To determine permanent and transitory data, we compute the *data lifespan* metric that we define as the number of instructions executed from the first until the last use of this data. Data references are ranged within six classes (or intervals) (Figure 10): P is the class of pointer usages, WRW are the references to data used within a WRW pattern, LS100 is the class of the references to data living no more than one hundred of instructions in the application... LS+ is the class of references to data that remains alive more than 10000 instructions.

We first analyzed the patterns of referencing data in the applications from MiBench, looking at the source code from each domain, and then patterns are captured through the automatic analysis we performed. The routine we link with calvin2+DICE simulator proceeds as follows: (i) memory locations written before being used are those used in a WRW pattern (ii) to detect pointers, we check if the register used to load data has been written before (memory address) (iii) for the other load operations, we evaluate the data lifespan metric.

In basicmath, data are used in a flow pattern, so are to be loaded in an auxiliary cache. bitcount's inputs are used in a set of computations during all the application time and must be cached. Image pixels in the process of susan image recognition are referenced over a large number of iterations. The process uses pointers to some image zones, while smoothing or detecting corners. susan_smoothing processes the image in a flow pattern, by sub-matrices 3x3.

The JPEG coder/decoder processes the input file, line by line. lame application manipulates complex structures via pointers, so it would be more appropriate to load in advance data pointers in the cache. TIFF benchmarks use their inputs in different ways. In tiff2bw, inputs are treated by buffer. This has not been detected in our profiling because of passing the read buffers as parameters to called functions. The image in tiffdither and tiffmedian is processed by set of two lines per iteration (pointed to by *thisline* and *nextline*). Whereas, each iteration of tiff2rgba main process uses the whole, or a significant part, of the image. Consequently, inputs processed in tiff2bw, tiffdither, and tiffmedian are to be loaded in an auxiliary cache, whereas it seems to be more appropriate to load the tiff2rgba processed image lines in the main cache.

At each iteration of its main loop, the stringsearch application reads a word and search for it in a sentence. These data are transitory and it is sufficient to load them in the auxiliary cache. Strings are manipulated using pointers that must be cached.

The network applications make use of node structures to implement routine tables (patricia) or the network graph (dijkstra). These structures are manipulated using pointers (10% of references). Dijkstra application uses the input file to fill the adjacency matrix, and use other structures in a WRW pattern. In patricia benchmark, the transitory data (used in a flow pattern) is IP addresses. At each iteration, patricia searches the path for one address using the routing tables, consequently it is effective to load the IP addresses in an auxiliary cache and to load the routine tables and the pointers in the main cache.

blowfish processes the inputs in a flow pattern, while the encryption/decryption key constitutes permanent data to be kept in the main cache during all of the application time. sha uses the *sha_info* structure data to cumulate the information collected from the processing of all the input buffers. Consequently, we have to load input buffers in the auxiliary cache and keep the *sha_info* in the main cache.

The adpcm coder/decoder processes one buffer of data at each iteration. This has not been detected in our simulation because of passing parameters by calls to the main methods of coding/decoding. crc32 application collects the information over the process iterations. fft and fft invers transform applications make use of some arrays in a WRW/permanent patterns. We can distinguish different patterns in gsm coder/decoder, in which data is transitory or permanent.

4.5. Synthesis

The characterization of MiBench applications shows that most of the studied kernels are quite regular and may be amenable to efficient performance on a decoupled architecture. However, the study of control flow regularity shows that on some of the applications, mispredicted conditional branches caused by some control/memory dependency are more frequent and may result on considerable loss of decoupling. In addition, we identify three categories of data whose use can cause some LOD, which are: permanent data, pointers, and transitory data.

In an embedded processor, these points could be ad-

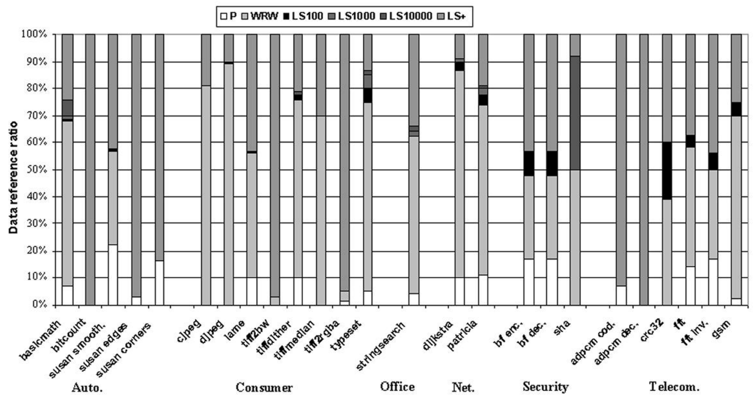


Figure 10. **Ratio of references to MiBench's data in each DRP.**

dressed through the use of different caches, for the different data categories. For example, we can use a large cache that receives control/permanent data and pointers, and an auxiliary one for transitory data.

5. Conclusion

In this paper, we addressed the adequation of the decoupled architecture to embedded applications. Decoupled architecture is known to work very efficiently whenever the execution does not suffer from inter-processor dependencies causing some loss of decoupling, called LOD events. In this study, we have analyzed the characteristics of a wide spectrum of embedded applications and address regularity of codes in terms of the LOD events that may occur. Three aspects of regularity have been addressed: control regularity, control/memory dependency, and patterns of referencing memory data.

The workload evaluation shows that most of kernels in MiBench may be amenable to efficient performance on a decoupled architecture. However, the study of control flow regularity shows that on some of the applications, mispredicted conditional branches caused by some control/memory dependency are more frequent and may result on considerable loss of decoupling. In addition, we identified three categories of data whose use can cause some LOD, which are: permanent data, pointers, and transitory data.

In an embedded processor, we feel that these points could be addressed through the use of different caches, for the different data categories. Selection of the cache target can be addressed either through the ISA or through the data layout. In future work, we will explore these directions in order to efficiently design a decoupled processor for embedded applications.

References

[1] Guthaus, M. R., Ringenberg, J. S., Ernst, D., Austin, T. M., Mudge, T., and Brown, R. B. , MiBench : A free, commercially representative embedded benchmark suite, In IEEE 4th Annual Workshop on Workload Characterization, Austin, TX, Dec. 2001.

[2] Lafage, T., and Seznec, A. , Choosing representative slices of program execution for microarchitecture simulations : A preliminary application to the data stream, In Workload Characterization of Emerging Applications, Kluwer Academic Publishers, Sept. 2000.

[3] EDN Embedded Microprocessor Benchmark Consortium, http://www.eembc.org.

[4] Bird, P., Rawsthrone, A., and Topham, N. P. , The effectiveness of decoupling, In proceedings of the 7th ACM International Conference on Supercomputing, 1993, pp. 47-56.

[5] Seznec, A., Felix, S., Krishnan, V., and Sazeides, Y. , Design tradeoffs for the alpha EV8 conditional branch predictor, In Proceedings of the 29th International Symposium on Computer Architecture (ISCA-02), New York, 2002, pp. 295–306.

[6] Smith, J.E., et al. , The ZS-1 central processor, In Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems, Palo Alto, CA, 1987, pp. 199-204.

[7] Wulf, W.A. , Evaluation of the WM architecture, In Proceedings of the International Symposium on Computer Architecture, Gold Coast, Australia, May 1992, pp. 382-390.

[8] Talla, D., and John, L. K. , MediaBreeze : a decoupled architecture for accelerating multimedia applications, ACM Computer Architecture News, ACM Press, ISSN 0163-5964, vol 29, n. 5, Dec. 2001.

[9] Goodman, J. R., Hsieh, J. T., Liou, K., Plezkun, A. R., Schechter, P. B., and Young, H. C. , PIPE: A VLSI decoupled architecture, In Proceedings of the 12th Annual International Symposium on Computer Architecture, 1985, pages 20–27.