LETTER

Impacts of Compiler Optimizations on Address Bus Energy: An Empirical Study

Hiroyuki TOMIYAMA[†a)], ***Member***

SUMMARY Energy consumption is one of the most critical constraints in the design of portable embedded systems. This paper describes an empirical study about the impacts of compiler optimizations on the energy consumption of the address bus between processor and instruction memory. Experiments using a number of real-world applications are presented, and the results show that transitions on the instruction address bus can be significantly reduced (by 85% on the average) by the compiler optimizations together with bus encoding.

key words: *compiler optimization, embedded systems, low energy, bus encoding*

1. Introduction

In the design of portable embedded systems, it is crucial to minimize the energy consumption in order to keep the battery life long. Address buses between processor and instruction memory are often one of major sources of energy consumption due to their large load capacitance as well as frequent accesses. Since the bus energy is almost proportional to the number of transitions on the bus, various techniques for address bus encoding have been proposed so far in order to reduce the bus transitions. Most of encoding techniques proposed for instruction address buses utilize the sequential nature of instruction accesses, i.e., most accesses are sequential with a fixed stride value. Examples of such encoding techniques include *Gray Coding* [1], *T0 Coding* [2], [3] and refinements of T0 such as *Inc-Xor* [4] and *T0-C* [5]. For sequential accesses, only one bit changes with the Gray encode, while no bit changes with the T0-based codes.

These encoding techniques rely on the sequential nature of instruction accesses. In general, however, not all of the instruction accesses are sequential due to control-flow instructions such as branches, jumps and function calls. It is known that the control-flow instructions typically account for 10 to 20% in program code [6]. In terms of address bus energy, it can be easily imagined that the straight-line code is more efficient than one with a lot of control-flow instructions. This brings the idea of transforming programs in such a way that the control flow is serialized. Some traditional compiler optimizations such as loop unrolling and function inlining serialize the control flow, thus they can be considered effective for bus energy reduction.

Manuscript received January 7, 2004.
Manuscript revised March 7, 2004.
Final manuscript received June 16, 2004.

[†]The author is with the Department of Information Engineering, the Graduate School of Information Science, Nagoya University, Nagoya-shi, 464-8603 Japan.
a) E-mail: hiroyuki@acm.org

This paper investigates the impacts of compiler optimizations on the energy consumption of instruction address buses. In this paper, we do not propose a new technique for compiler optimization or bus encoding. The contribution of this paper is to conduct experiments with a number of realistic benchmark programs and demonstrate that compiler optimizations have large impacts on the instruction address bus energy. To the best of our knowledge, no previous literature explicitly studied this topic, and this is the first paper which studies it and presents extensive experiments. In [7] and [8], data layout techniques for energy minimization of address buses were investigated in the fields of high-level synthesis and compilation, respectively. However, both of them focused on address buses for data memory. This paper, on the contrary, addresses energy minimization of address buses for instruction memory.

This paper is organized as follows. Section 2 describes our experimental procedure and bus encoding and compiler optimization techniques used in the experiments. In Sect. 3, experimental results are presented. Finally, Sect. 4 concludes this paper with a summary and future directions.

2. Experimental Setup

We have conducted a set of experiments to test the effectiveness of compiler optimizations for address bus energy minimization. This section describes the bus encoding techniques and the compiler optimization techniques used in our experiments as well as the experimental procedure.

2.1 Experimental Procedure

The flow of our experiments is shown in Fig. 1. As benchmark programs, we selected 10 real-world media applications from the *MediaBench* suite [12] including MPEG, JPEG and so on. Each program was compiled with GNU C Compiler 2.7.2.3 to generate the object code. At that time, several optimization techniques were applied. Then, the object code was executed on the *SimpleScalar* processor simulator [14] to obtain a trace of instruction addresses. Finally, the address trace was fed by *bus transition analyzer* which we developed. The analyzer involves four types of address bus model, i.e., a normal bus without encoding, the T0-encoded bus, the T0-C bus, and the Inc-Xor bus, and it reports the number of bus transitions for each bus model. This procedure was repeated many times with changing compiler optimization options as well as benchmark programs.

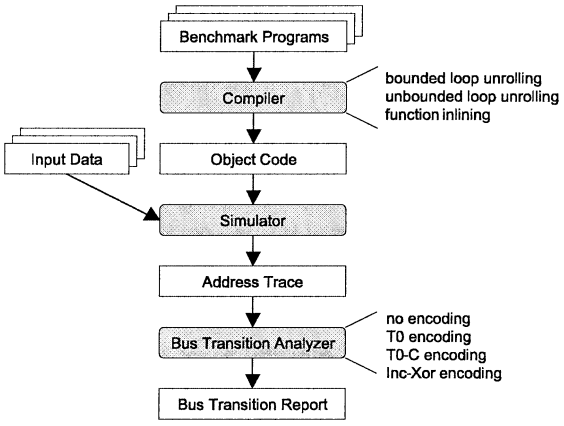

Fig. 1 Flow of our experiments.

In the rest of this section, we describe in more detail the bus encoding techniques and compiler optimizations used in our experiments.

2.2 Address Bus Encoding Techniques

In our experiments, three address bus encoding techniques, i.e., *T0*, *T0-C* and *Inc-Xor*, were used. We use the following notations for explanation of the three encoding techniques.

`b(t)`: Original address value to be sent at time `t`.
`B(t)`: Encoded value which is actually transfered on the bus at time `t`.
`S`: Stride value, i.e., the difference between consecutive addresses.

T0 is a redundant code developed by Benini et al. for instruction address buses [2], [3]. In addition to the address lines, it has a redundant bit line, named `INC`. For sequential accesses, the sender sets the `INC` line and keeps the address lines unchanged. Otherwise, the original address value $b(t)$ is sent on the bus and `INC` is de-asserted. A pseudo-code of the T0 encoder is given below.

```
if (b(t) == b(t-1) + S) {
   B(t) = B(t-1);
   INC = 1;
} else {
   B(t) = b(t);
   INC = 0;
}
```

The *T0-Concise* code (or *T0-C* in short) developed by Aghaghiri et al. [5] is an irredundant code based on T0. The T0-C encoder works as follows.

```
if (b(t) == b(t-1) + S) {
   B(t) = B(t-1);
} else if (B(t-1) != b(t)) {
   B(t) = b(t);
} else {
   B(t) = b(t-1) + S;
}
```

The last case, i.e., `(b(t) != b(t-1) + 1) and (B(t-1) == b(t))`, can be considered as an exceptional case to the T0 code. Since such a case rarely happens in practice, T0-C is more efficient than T0 in terms of bus transitions due to its irredundancy.

In [4], Ramprasad et al. developed another T0-based irredundant code, called *Inc-Xor*[†]. The encoder works as follows.

```
B(t) = b(t) ⊕ (b(t-1) + S) ⊕ B(t-1)
```

Here, $\oplus$ denotes the Exclusive-Or (Xor) function. It is easily observed that no bit changes when the accesses are sequential.

The experimental results in [5] show that T0, T0-C and Inc-Xor achieve the average savings of bus activities by 62.0%, 73.1% and 75.0%, respectively. The experiments in [4] also demonstrate the effectiveness of Inc-Xor over the T0 and Gray codes.

Several other techniques have been proposed, for example in [9], [10] and [11]. They are effective not only for instruction address buses but also data address buses, but the hardware overhead required by them is not negligible. In our work, we use the three T0-based coding techniques mentioned above because of their simplicity and effectiveness.

2.3 Compiler Optimizations

In this work, the goal of compiler optimization is to serialize instruction accesses. For this purpose, we tested two compiler optimization techniques, *loop unrolling* and *function inlining*.

Loop unrolling is one of widely used compiler optimizations which makes multiple copies of the loop body and reduces the number of iterations of the loop. If the number of iterations is known at compile-time, such a loop is called a *bounded loop*, and unrolling of bounded loops is called *bounded loop unrolling*. Otherwise, it is called *unbounded loop unrolling*. Traditionally, loop unrolling is used to improve the performance. The performance improvement comes from two reasons. One is that it reduces the number of executions of condition testing, the corresponding branch, and loop index variable updates. The other is that loop unrolling extends opportunities for other compiler optimizations such as instruction-level parallelizing scheduling. It is obvious that loop unrolling is effective for reducing the address bus energy as well as enhancing the performance since the number of branches is decreased. Since loop unrolling generally increases code size, however, it is generally unreasonable to unroll large loops. Most compilers have their own heuristics to select loops to be unrolled.

Function inlining, which replaces a function call with the body of the function, is also one of the most widely used compiler optimizations. It has several benefits as follows.

[†]In [13], Inc-Xor is refered as *T0-Xor*.

First, it eliminates the function call overhead (extra code) for storing and restoring registers and the program counter, adjusting stack and frame pointers, jumping to the function, and so on. Second, function inlining extends opportunities for other compiler optimizations such as constant propagation, dead-code elimination, control flow simplification, and so on, which are usually done on a function-by-function basis. Obviously, function inlining is also effective for reducing address bus energy since it eliminates function calls which disturb sequential accesses. However, function inlining often increases the code size, especially in case the function is called from more than one point of the program code. Therefore, inlining large functions is unreasonable. Most compilers have their own heuristics to select functions to be inlined.

Loop unrolling and function inlining can be applied separately as well as together. In our experiments, we tested the following combinations of compiler optimization:

- `O2`: Compiler optimizations which do not increase the code size (i.e., enabled by GCC's "`-O2`" option).
- `O2+FI`: Function inlining in addition to `O2`.
- `O2+BLU`: Bounded loop unrolling in addition to `O2`.
- `O2+BLU+ULU`: Unbounded loop unrolling in addition to `O2+BLU`.
- `O2+FI+BLU+ULU`: All of the above optimizations.

3. Experimental Results

Figures 2(a)–(k) show the results of our experiments. For each benchmark program, the number of transitions on the instruction address buses is depicted where the baseline is `O2` without bus encoding. Similar to the results in [5], Inc-Xor was the best bus encoding for most cases in our experiments. In case of the `O2` compiler optimization, Inc-Xor achieved over 80% saving of bus transitions on the average. When all of the compiler optimizations were applied, i.e., `O2+FI+BLU+ULU`, the bus transitions was saved by 83%.

From Fig. 2, we can see that applying all the compiler optimizations did not always generate the most energy-efficient code. In most cases, function inlining and loop unrolling contributed to reduction in bus transitions. For `g721encode`, `g721decode` and `mpeg-decode`, bus transitions were significantly reduced by the compiler optimizations with the Inc-Xor encoding. In some cases such as

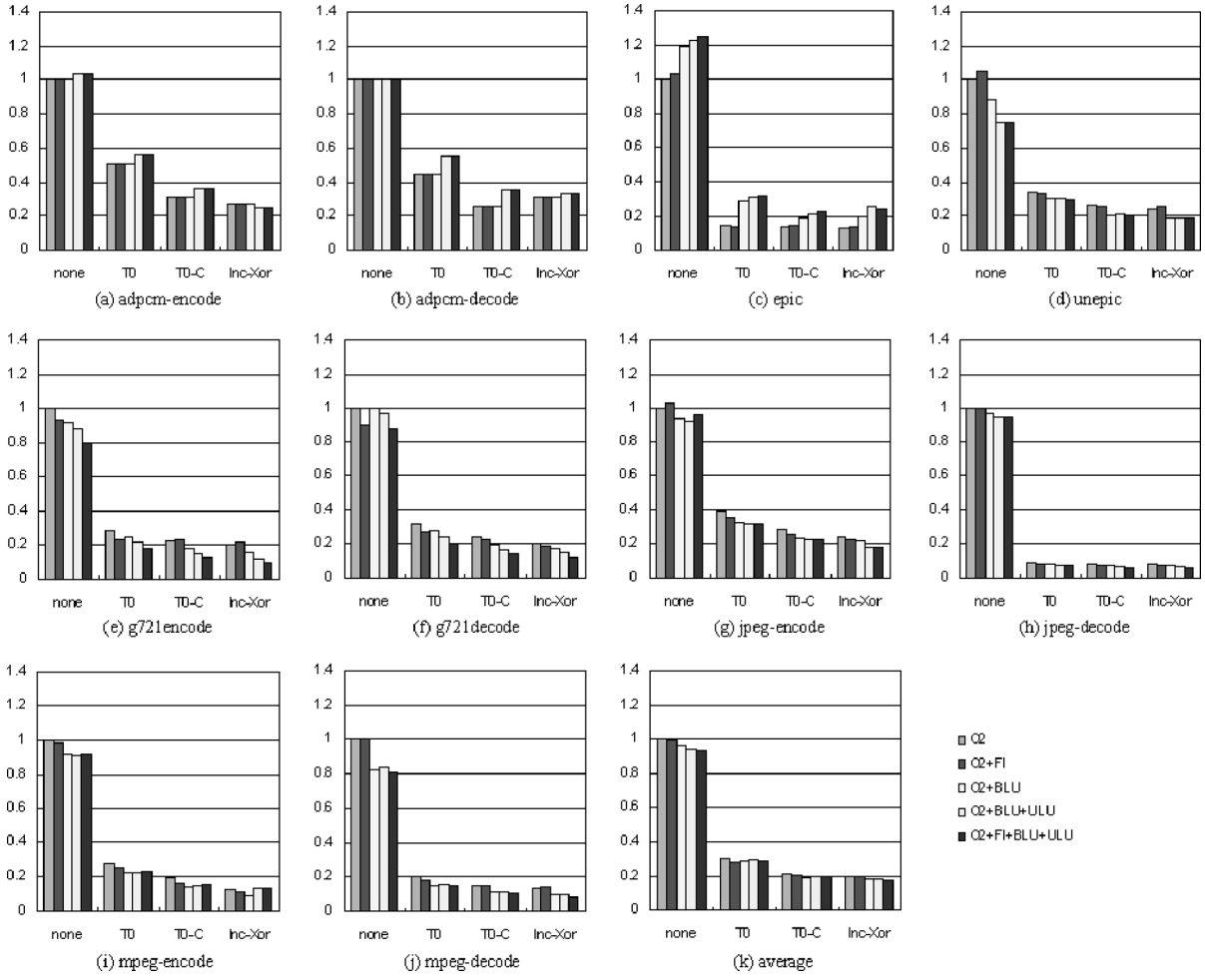

Fig. 2 The number of transitions on instruction address bus.

adpcm-decode and epic, however, the compiler optimizations resulted in increased transitions on the contrary. For mpeg-encode, bounded loop unrolling was very effective, but function inlining and unbounded loop unrolling were not.

Thus, the effectiveness of the compiler optimizations for bus energy varies among the programs. In the design of embedded systems, therefore, we need to apply the compiler optimizations very carefully. At present, it is very difficult to know which compiler optimization is effective for a given program *before* we apply it. A simple approach is to repeat compilation and simulation with changing compiler optimization options in order to obtain the most energy-efficient code. Figure 3 shows the best result among the five optimization options (i.e., O2, O2+FI, O2+BLU, O2+BLU+ULU and O2+FI+BLU+ULU). Inc-Xor was used for bus encoding. The figure indicates the normalized number of bus transitions where the baseline is O2. In our results, the O2 option was the best optimization option for adpcm-decode and epic, O2+BLU+ULU for unpic, O2+BLU for mpeg-encode, and O2+FI+BLU+ULU for the other programs. Figure 3 demonstrates that judicious application of the compiler optimizations reduces bus transitions by 24% on the average. As mentioned above, the Inc-Xor encoding is assumed in Fig. 3, and Fig. 2(k) shows that Inc-Xor achieves an average reduction of 80% in bus transitions. Hence, the average saving of 85% can be obtained by the compiler optimizations together with bus encoding, compared with the O2 compiler option without bus encoding.

Figure 2 shows that for many programs the number of bus transitions was reduced by function inlining and loop unrolling even when the address bus was not encoded. This is because the number of instructions executed was reduced by the optimizations[†]. In summary, there exist two reasons why function inlining and loop unrolling reduces transitions on the instruction address bus. One reason is that they serialize the program control flow, and the other one is that they reduce the number of instruction memory accesses. This is demonstrated by Figs. 4 and 5. Figure 4 shows the normalized number of instructions executed where the baseline is O2. O2+FI+BLU+ULU achieved 9.5% reduction in the number of instructions (i.e., 9.5% performance improvement) on the average. Figure 5 shows the percentage of nonsequential instruction accesses. With the O2 compiler option, the ratio of nonsequential accesses was 13.3% on the average. With the O2+FI+BLU+ULU compiler option, the ratio went down to 12.0%.

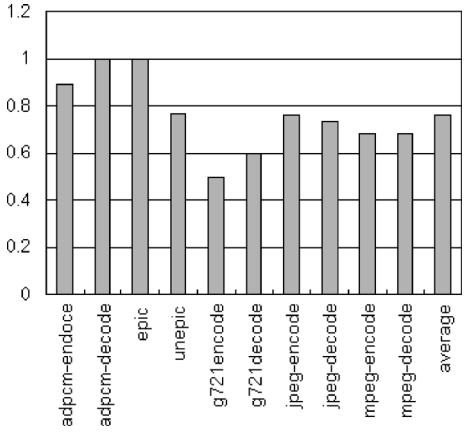

Fig. 3 The effectiveness of compiler optimization exploration.

Although function inlining and loop unrolling are effective to reduce bus transitions, they have a common drawback, i.e., the code size problem, as mentioned in the previous section. In Fig. 6, the code size for each compiler option is presented. For O2+FI+BLU+ULU, the code size was increased by 7.3% on the average. It is observed that the increase in code size varies among the programs. For example, the increase in code size was very small for g721encode and g721decode, but the bus transitions were significantly reduced. However, in most cases, there is a trade-off between address bus energy and code size, so it is very important to explore and find the optimal trade-off point for each application.

So far, we do not assume memory hierarchy. If the instruction cache exists, loop unrolling and function inlining

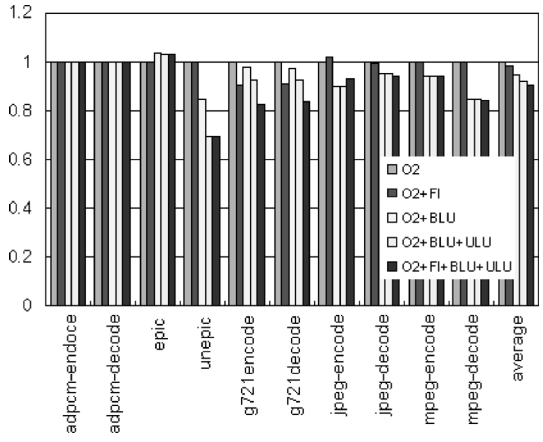

Fig. 4 The number of instructions executed.

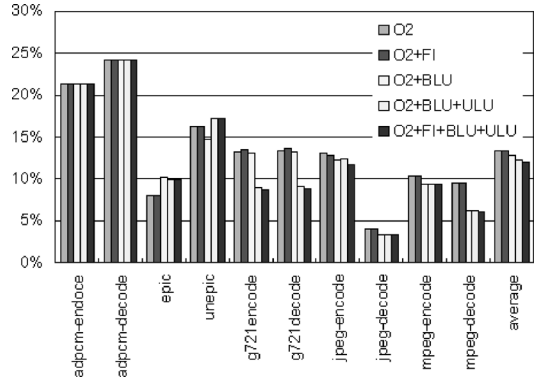

Fig. 5 Percentage of nonsequential instruction accesses.

[†]Recall that function inlining and loop unrolling were originally developed to reduce the number of instructions to be executed.

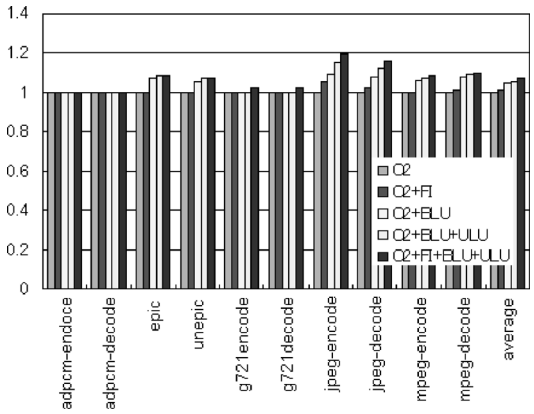

Fig. 6 Increase in code size.

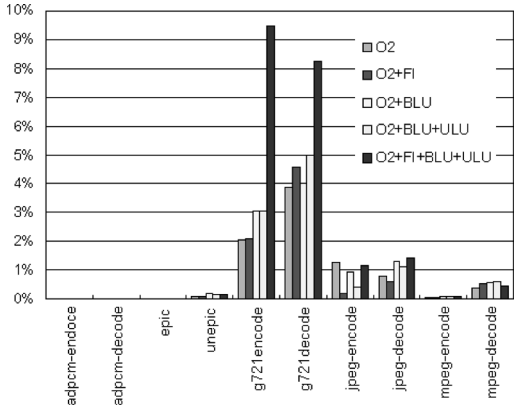

Fig. 7 Instruction cache miss rates. (8 K-byte direct-mapped cache)

reduce the energy of the address bus between the processor and the instruction cache. However, the increased code size due to the compiler optimizations often leads to degradation of the cache performance, which requires extra energy for accessing main memory. In order to study the cache effect, we have run cache simulation. Figure 7 shows the miss rates of the instruction cache across the benchmark programs. For `g721encode` and `g721decode`, the cache performance was seriously degraded which may offset the energy saving on the address bus. On the other hand, the cache performance was improved for `jpeg-encode` in despite of the increased code size. It is also seen that function inlining reduces cache misses for `jpeg-decode`. This is mainly because function inlining often improves the spatial locality of the program code. For some other benchmark programs, no significant difference in the cache performance was observed. Thus, these results show that the impacts of the compiler optimizations on the cache performance is largely dependent on the program.

As we have seen above, the compiler optimizations produce various side effects on performance and energy of processor and memory system as well as code size, and these effects are sometimes good but sometimes bad depending on the program. Development of a systematic methodology to apply the compiler optimizations with considering all the effects is one of our remaining work.

4. Conclusions

This paper studied the impacts of compiler optimizations, i.e., function inlining and loop unrolling, on the energy consumption of the address bus between processor and instruction memory. We conducted a set of experiments using real-world applications. The experimental results showed that transitions on the instruction address bus were redeced by 85% by applying the compiler optimizations together with bus encoding.

Since the compiler optimizations produce various side effects, we are currently developing a systematic methodology for low-energy compilation which takes into account these side effects.

Acknowledgment

The author would like to thank Professor Nikil Dutt of University of California at Irvine and Professor Hiroaki Takada of Nagoya University for their valuable suggestions. This work was partially supported by JSPS Grant-in-Aid for Young Scientists (B) #16700058.

References

[1] C.L. Su, C.Y. Tsui, and A.M. Despain, "Saving power in the control path of embedded processors," IEEE Des. Test Comput., vol.11, no.4, pp.24–31, 1994.

[2] L. Benini, G. De Micheli, E. Macii, D. Sciuto, and C. Silvano, "Asymptotic zero-transition activity encoding for address buses in low-power microprocessor-based systems," Proc. Great Lakes Symp. on VLSI, pp.77–82, 1997.

[3] L. Benini, G. De Micheli, E. Macii, D. Sciuto, and C. Silvano, "Address bus encoding techniques for system-level power optimization," Proc. Design Automation and Test in Europe, pp.861–866, 1998.

[4] S. Ramprasad, N.R. Shanbhag, and I.N. Hajj, "A coding framework for low-power address and data busses," IEEE Trans. Very Large Scale Integr. (VLSI) Syst., vol.7, no.2, pp.212–221, 1999.

[5] Y. Aghaghiri, F. Fallah, and M. Pedram, "Irredundant address bus encoding for low power," Proc. Int'l Symp. on Low-Power Electronics and Design, pp.182–187, 2001.

[6] J.L. Hennessy and D.A. Patterson, Computer Architecture: A Quantitative Approach, Morgan Kaufmann Publishers, 2nd ed., 1996.

[7] P.R. Panda and N.D. Dutt, "Low-power memory mapping through reducing address bus activity," IEEE Trans. Very Large Scale Integr. (VLSI) Syst., vol.7, no.3, pp.309–320, 1999.

[8] H. Tomiyama, H. Takada, and N. Dutt, "Memory data organization for low-energy address buses," IEICE Trans. Electron., vol.E87-C, no.4, pp.606–612, April 2004.

[9] E. Musoll, T. Lang, and J. Cortadella, "Working-zone encoding for reducing the energy in microprocessor address buses," IEEE Trans. Very Large Scale Integr. (VLSI) Syst., vol.6, no.4, pp.568–572, 1998.

[10] L. Benini, G. De Micheli, E. Macii, M. Poncino, and S. Quer, "Power optimization of core-based systems by address bus encoding," IEEE Trans. Very Large Scale Integr. (VLSI) Syst., vol.6, no.4, pp.554–562, 1998.

[11] M.N. Mamidipaka, D.S. Hirschberg, and N. Dutt, "Adaptive low-power address encoding techniques using self-organizing lists," IEEE Trans. Very Large Scale Integr. (VLSI) Syst., vol.11, no.5, pp.827–834, 2003.

[12] C. Lee, M. Potkonjak, and W.H. Mangione-Smith, "MediaBench: A tool for evaluating and synthesizing multimedia and communicatons systems," Prof. Int'l Symp. on Microarchitecture, pp.330–335, 1997.
[13] W. Fornaciari, M. Polentarutti, D. Sciuto, and C. Silvano, "Power optimization of system level buses based on software profiling," Proc. Int'l Workshop on Hardware/Software Codesign, pp.29–33, 2000.
[14] SimpleScalar Tools, http://www.simplescalar.com/