Application-Specific Instruction Sets Processor with Implicit Registers to Improve Register Bandwidth

Ginhsuan Li, Chiuyun Hung, Desheng Chen, and Yiwen Wang

***Abstract*—**Application-Specific Instruction (ASI) set Processors (ASIP) have become an important design choice for embedded systems due to runtime flexibility, which cannot be provided by custom ASIC solutions. One major bottleneck in maximizing ASIP performance is the limitation on the data bandwidth between the General Purpose Register File (GPRF) and ASIs. This paper presents the Implicit Registers (IRs) to provide the desirable data bandwidth. An ASI Input/Output model is proposed to formulate the overheads of the additional data transfer between the GPRF and IRs, therefore, an IRs allocation algorithm is used to achieve the better performance by minimizing the number of extra data transfer instructions. The experiment results show an up to 3.33x speedup compared to the results without using IRs.

***Keywords*—**Application-Specific Instruction-set Processors, data bandwidth, configurable processor, implicit register.

I. Introduction

ASIPs provide a compromise between custom designs and general purpose processors. A base processor with a basic instruction set is augmented with application-specific functional unit (AFU) that implements application-specific instruction-set (ASI) extensions for complex processing tasks as either single-cycle (combinatorial) or multi-cycle (sequential) operations. The control-flow within the application is directed by the base processor, whereas computation intensive regions are implemented as custom logic. A dedicated link between custom logic and the base processor provides an efficient communication interface. Reusing a pre-verified, pre-optimized base processor reduces the design complexity, and the time to market.

Among the best known examples of extensible ASIPs are CoWare [1], Tensilica Xtensa [2] and Altera Nios/Nios II [3], and some levels of customizability have also been added on traditional well-established architectures such as MIPS CorExtend [4] or PowerPC APUs [5]. The research community has expended a considerable amount of effort in the ASIP area for almost a decade. The issues involved in the ASIP design were surveyed in [6]. The ASIP architecture and the compiler co-exploration problem are addressed in [7].

Application specific instruction set processor problem is defined as a process to automatically generate ASIs from an application in order to meet certain design objectives. An existing work in ASIP generally consists of three steps. 1) template generation, 2) ASIs selection and 3) application replacement. Template generation can be loosely described as a process of identifying a subgraph from the application data-flow graph (DFG) to form a single ASI in order to maximize some metrics (typically performance). This step generates a set of templates, which will be evaluated for ASIs implementation. ASIs selection evaluates the templates in terms of their performance, area, or power and selects a subset of them that meets the design constraints.

In this work, we apply formal optimization techniques to generate ASIs from C code. We target architectures where the data bandwidth between the base processor and the custom logic is constrained by the available GPRF ports in Fig. 1.

Our method is applicable to architectures where the data bandwidth is limited by dedicated data transfer channels. Given the available data bandwidth, our approach identifies the most profitable ASIs based on a heuristic algorithm. The data transfer overhead when generating and evaluating ASIs is explicitly considered. We demonstrate our automatically customized processor within the silicon area using FPGA synthesis results.

The contribution of the current paper is that we provide an ASI input/output model which considers the I/O abstraction with the base processor GPRF bandwidth constraints and extra data transfer costs. Another contribution of this paper is that the model and a heuristic algorithm are successfully integrated into the ASIP design flow to automatically generate ASIs from C codes.

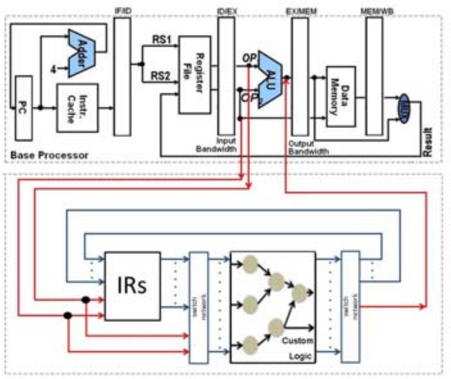


Fig. 1 The Datapath of the application-specific instruction processor (data bandwidth is limited by the GPRF I/O ports)

II. Related Work

Existing approaches attempt to discover the candidate application specific instructions by exploiting the observation

Ginhsuan Li[1], Chiuyun Hung, Desheng Chen, and Yiwen Wang is with the Information Engineering and Computer Science Department of Feng Chia University, Taichung, R.O.C., (e-mail: pipitra1759@gmail.com[1], p9431850, dschen and ywang@fcu.edu.tw).

that recurring subgraphs frequently exist in the dataflow graphs (DFG) of applications, and then to select an appropriate subset of the candidate instructions to maximize performance under certain architectural constraints (e.g., the number of input and output operands, area constraints, etc.). By implementing these frequently occurring subgraphs in hardware as instructions, performance improves, code size decreases, and energy consumption is reduced. In template generation, most approaches use the number of I/O ports of the register file to constrain the set of subgraphs that can be enumerated [8]–[14]. In candidate selection, the problem of individual templates appeared in multiple candidates make selection more difficult. Selecting candidates with a given area constraint is similar to the 0/1 knapsack problem [15]. It is widely known that the 0/1 knapsack problem is NP-complete. Strategies are needed to avoid intractability in this step for design automation.

Most prior techniques for ASI generation use the number of I/O ports of the register file to constrain the set of subgraphs that can be enumerated; this yielded effective pruning criteria that reduced the size of the search space for ASI identification. Although the existing techniques are efficient in identifying the promising candidate instructions, [16] points out that most of the speedup (about 60%) comes from the cluster with more than two input operands. This exceeds the number of read ports available on the register file of a typical embedded RISC processor core. Strictly following the two-input single-output constraint, generally leads to small clusters with limited speedup.

Generation of larger clusters with extra inputs is allowed in [17] by using the custom-defined state registers to store the additional operands. Unfortunately, at least one extra cycle is needed for each additional input to be loaded into a custom-defined state register. The communication overhead incurred because of these data transfers between the core processor, and the custom logic can significantly offset the gain from forming a large cluster. A multi-ported register file can increase the data bandwidth. However, additional read and write ports result in power consumption and cycle time. The Tensilica Xtensa uses state registers to explicitly move additional input and output operands between the base processor and custom units. Clever binding of base processor registers to state registers at compile time reduces the number of data transfers. In addition, state register approach solves the problem of encoding many operands within a fixed length instruction word. Cong *et al.* [18] have presented a hash-mapped low-cost architectural extension and associated internal register binding compilation techniques to efficiently reduce the communication overhead due to data transfers between the core processor and the AFU. However, extra hash table increases AFU area overhead. Pozzi et al. [19] reduce the data transfer overhead by overlapping execution cycles with data transfer cycles for pipelined multi-cycle ASIs.

We integrate the data bandwidth information directly into the optimization process, and we explicitly account for the cost of the data transfers between base register file and custom implicit registers as part of the optimization. Since our formulation can take advantage of the increased data bandwidth, the approach of Pozzi et al. [19] can be combined with ours to further improve the performance of multi-cycle ASIs.

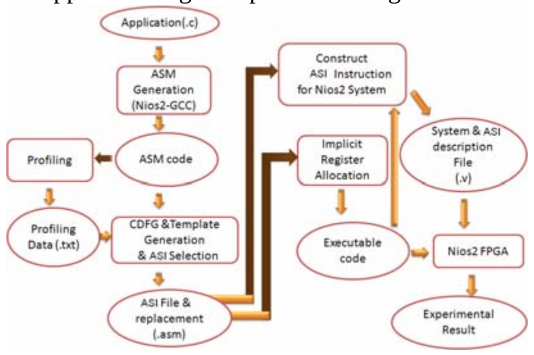
III. THE OVERALL DESIGN FLOW

The Altera NiosII is selected as a base processor to implement the target ASIP but not limited to this specific architecture. The input of the proposed design flow is application specific programs in C code. The gcc tool chain is used to obtain the profiling information for each basic block in the application programs which is corresponded to the occurrence of each primitive base processor assembly instruction. Therefore, the control/data flow graphs (CDFGs) of the entire application program are generated for the analysis of the data dependencies among the primitive instructions.

The subgraphs of the CDFGs are enumerated as ASI templates and then the structural equivalent templates within isomorphism classes will be grouped as ASI candidates. For each ASI candidate, the corresponding behavioral hardware descriptions are implemented in Verilog as well as be synthesized via Altera Quartus II and SOPC builder to estimate the hardware area cost.

The most profitable candidates will be selected according to the proposed heuristic algorithm to balance the time-area design constraints. The selected ASI candidates will be used to conduct the graph covering on the CDFGs based on a proposed heuristic algorithm. The matching code segments on the CDFGs will be replaced with new opcodes representing the ASIs.

Once the most profitable candidates are selected, we replace the matching code segments with an opcode representing the new instruction. Finally, the ASIP and the corresponding application codes with ASIs are verified on the Altera DE2-70 board to demonstrate the correctness and performance of the proposed approach. Fig. 2 depicts our design flow.


Application(.c)
ASM Generation (Nios2-GCC)
ASM code
Profiling
Profiling Data (.txt)
CDFG &Template Generation & ASI Selection
ASI File & replacement (.asm)
Construct ASI Instruction for Nios2 System
Implicit Register Allocation
Executable code
System & ASI description File (.v)
Nios2 FPGA
Experimental Result
Fig. 2 The proposed design flow

IV. PROPOSED APPROACH

A. Problem Formulation

A basic block is represented using a directed acyclic graph G(V, E) where nodes V represent operations, edges E represent register dependencies between operations. A template T is an induced subgraph of G. A template is convex if there exists no path in G from a node $u \in T$ to another node $v \in T$ which involves a node $w \notin T$. The convexity constraint is imposed on the templates to ensure that no cyclic dependencies are introduced in G, and a feasible schedule can be generated.

We estimate the software cost of a template, SW(T), as the sum of the software latencies of the instructions contained in T. We estimate the cost of moving T to a custom datapath as the sum of estimated hardware execution latency, HW(T).

B. Template and Candidate Generation

Our template generation algorithm iteratively exploits valid subgraphs in order to generate a set of templates. A valid subgraph which must satisfied two constraints. First, there are no load, store, jump or branch instructions include in a subgraph. Second, for a given application basic block, the. Template generation algorithm is applied on all basic blocks, and a set of application-specific instructions templates are generated.

After template generation is done, we calculate the isomorphic classes; the set of generated templates is partitioned into C_k different isomorphic classes.

C. Calculation of input and output data transfers

We assume N_{in} read ports, and N_{out} write ports supported by the base register file. If the number of inputs for a template is larger than N_{in}, we assume additional data transfers from the base register file to custom implicit registers. If the number of outputs for a template is larger than N_{out}, we assume additional data transfers from custom implicit registers to the base register file.

We introduce an integer variable I(T) to indicate the number of inputs for a template T. An input operand $e \in E$ in the basic block is an input of the template T if it has at least one immediate successor in T. We calculate the number of additional data transfers from base register file to the custom logic as I_Penalty(T):

$$\text{I\_Penalty}(T) = \left\lceil \frac{I(T)}{N_{in}} \right\rceil - 1 \quad (1)$$

We introduce an integer variable O(T) to indicate the number of outputs for a template T. A node $v \in V$ generates an output operand of T if it is in T, and it has at least one immediate successor not in T, then the output operand is an output for T. We calculate the number of additional data transfers from the custom logic to the base register file as O_Penalty(T):

$$\text{O\_Penalty}(T) = \left\lceil \frac{O(T)}{N_{out}} \right\rceil - 1 \quad (2)$$

The cycle saved CS(T) of the template T is defined as the value which estimates the reduction in the schedule length of the application by replacing the template with an ASI, multiplied by the occurrences Occ(T) of the template. Formally:

$$CS(T) = Occ(T) * (SW(T) - HW(T) - \text{I\_Penalty}(T) - \text{O\_Penalty}) \quad (3)$$

Take Fig. 3 for example. Assume the software latency of each node is 1 cycle, and the critical path latency is 1 cycle. Under an I/O constraint of 2/1, additional data transfer moving operand c and d into implicit register before template T is executed costs one cycle. Additional data transfer moving operand f back to base register file after template T is finished costs one cycle. Thus, the cycle saved by template T is 0 (3-1-1-1) cycle.

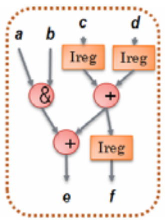


Fig. 3 An example shows the I_Penalty and O_Penalty

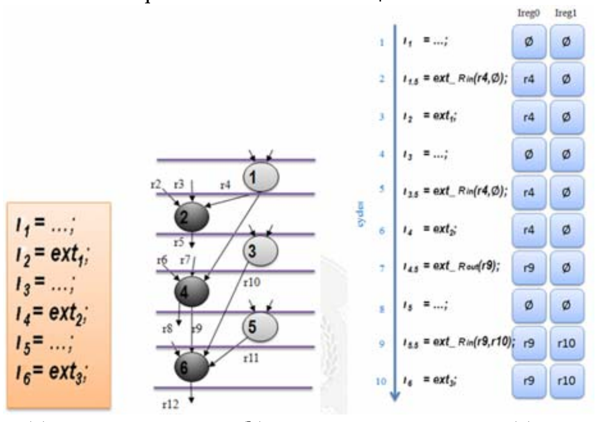
D. ASI Selection

A priority value PValue for each candidate is calculated and we select the candidate which has the highest priority value as an ASI. After an ASI is selected, PValue of all candidates are recalculated. Multiple ASIs are selected by repeating the above steps. The priority value for each candidate C_k is calculated as (4).

$$PValue(C_k) = \sum_{T \in C_k} CS(T) \quad (4)$$

E. Input and Output Operands Post Improvement

The primitive and application-specific instructions inside one basic block are ordered according to the instruction scheduling as shown in Fig. 4 (a). Fig. 4(b) shows a DFG example annotated with a sequence number for each instruction. Fig. 4 (c) shows a possible way to implement the ASIs. Suppose the register file has only two read ports, and all ASIs have more than two input operands, then one move ext_R_{in} instruction in cycle 2 will be required for instruction I_2 and one move ext_R_{in} in cycle 5 will be required for instruction I_4 and one move ext_R_{in} instruction in cycle 9 will be required for instruction I_6. Similarly, one more move ext_R_{out} instruction in cycle 7 will be required for instruction I_4.



```
I1 = ...;
I2 = ext1;
I3 = ...;
I4 = ext2;
I5 = ...;
I6 = ext3;
```

```
1   I1   = ...;
2   I1.5 = ext_Rin(r4,Ø);
3   I2   = ext1;
4   I3   = ...;
5   I3.5 = ext_Rin(r4,Ø);
6   I4   = ext2;
7   I4.5 = ext_Rout(r9);
8   I5   = ...;
9   I5.5 = ext_Rin(r9,r10);
10  I6   = ext3;
```

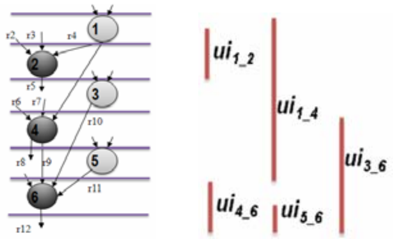
Ireg0	Ireg1
Ø	Ø
r4	Ø
r4	Ø
Ø	Ø
r4	Ø
r4	Ø
r9	Ø
Ø	Ø
r9	r10
r9	r10

(a) (b) (c)

Fig. 4(a) Assembly code. (b) The DFG of (a). (c) The implementation of ASIs

As mentioned earlier, if the operand number of an ASI exceeds the available read port count, extra data transfer (or ext_R_{in}) instructions are needed to copy operands from the base register file to the implicit register in the custom logic. In our proposed approach, if an operand is already in the implicit register, one move instruction can be saved.

The usage interval (ui) between primitive and ASI instructions is derived to record the implicit register usage. If variable r is assigned within the same basic block of the subject use, then the usage interval [p,c], where the sequence number of the assignment instruction is p and the sequence number of the use instruction is c as shown in Fig. 5.



(a) (b)

Fig. 5 (a) the DFG of assembly code (b) The usage interval

We propose a heuristic implicit register allocation algorithm. All ASIs will be sorted according to the decreasing sequence number. For the example in Fig. 5(a), the largest sequence number of ASI is I_6, if the $I(I_6)$ exceeds the number of base register file port, then we will check if any input operand $I(I_6)$ is the output operand from previous ASI. Thus, the ui[4,6] will occupy an implicit register (Ireg). The number of input operands of I6 becomes three, still larger than N_{in}. Next, the ui[3,6] will occupy an Ireg, however Ireg0 is occupied by other usage interval. The Ireg1 will be allocated for ui[3,6].

Subsequently, the ui[1,4] of instruction I_4 will be checked if it is overlapped with ui[4,6] of Ireg0. Since ui[1,4] and ui[4,6] are not overlapped, the Ireg0 will be allocated to ui[1,4]. Then, all input operands of instruction I_2 are ready. The final result of the implicit register allocation is shown in Fig. 6.

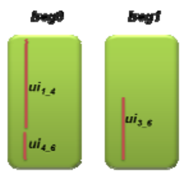
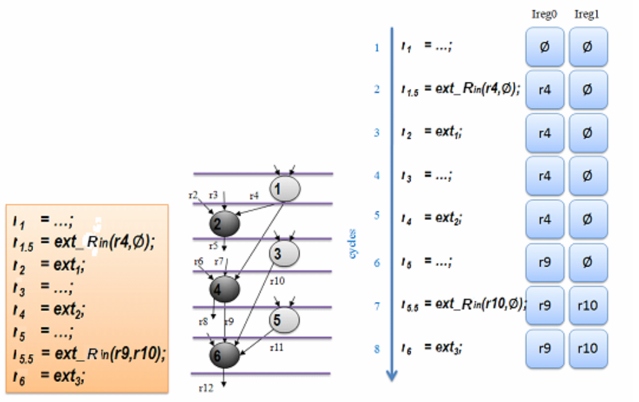


Fig. 6 Implicit register allocation result

After implicit register allocation, the number of additional data transfer can be reduced by reusing the operands between ASIs. Fig. 7(c) shows the input and output operands post improvement after implicit register allocation.



```
I1   = …;
I1.5 = ext_Rin(r4,Ø);
I2   = ext1;
I3   = …;
I4   = ext2;
I5   = …;
I5.5 = ext_Rin(r9,r10);
I6   = ext3;
```

```
1  I1   = …;                 Ø   Ø
2  I1.5 = ext_Rin(r4,Ø);     r4  Ø
3  I2   = ext1;              r4  Ø
4  I3   = …;                 r4  Ø
5  I4   = ext2;              r4  Ø
6  I5   = …;                 r9  Ø
7  I5.5 = ext_Rin(r10,Ø);    r9  r10
8  I6   = ext3;              r9  r10
```

(a) (b) (c)

Fig. 7 (a) The post improvement assembly code. (b) The DFG of (a). (c) Implementation of ASIs

V. Experimental Results

We evaluate our technique using Altera Nios II DE2-70 to estimate cycle counts, and hardware synthesis for exact timing and area information. The base register file supports two read ports and a single write port. We generate implicit registers for each application-specific instructions and move instructions that provide single cycle latency transfers between base register file and custom logic.

We apply our algorithms on five benchmarks: MM, Qsort, Dijkstra, SHA and AES form Mibench[20] and MP3[21]. Relaxation of input/output constraints results in coarser gain application-specific instructions (i.e., larger dataflow subgraphs). Such ASIs often offer higher speedup at the expense of higher area. In Fig. 8, we study the improvement in speedup using additional data transfers post improvement after implicit register allocation. Up to 3.33x speedup is reachable given 2 read and single write ports.

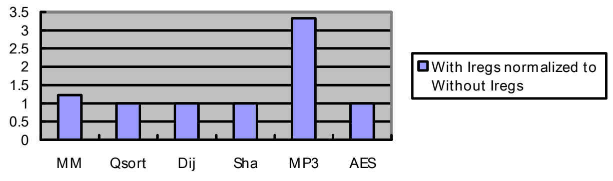


Fig. 8 Speedup improves with implicit registers comparison

Table I summarizes the hardware area for each generated ASIP. Table I shows that up to 20 % area overhead of the customized designs can obtain 3.33x speedup.

TABLE I
HARDWARE AREA

Benchmarks	Nios II/f LEs	Nios II/f + ASIs LEs	LEs overhead	# ASIs	MAX N_{in}/N_{out}	# Iregs
MM	3942	4282	8.6%	1	3/1	1
Qsort	3942	4173	5.9%	1	2/1	0
Dijkstra	3942	4358	10.6%	4	3/1	1
Sha	3942	4285	8.7%	2	4/1	2
MP3	3942	4698	19.1%	6	5/2	3
AES	3942	4318	9.5%	4	2/1	0

VI. Conclusion

Our approach is based on a heuristic algorithm that integrates the data bandwidth information into the ASI identification process. For an embedded processor with GPRF only two read ports and one write port, our methodology can minimize the potential additional data transfer instructions to achieve the performance up to 3.33x speedup with only 20% area overhead. The Altera Nios II is used as a base processor on the DE2-70 board to demonstrate the correctness and feasibility of the proposed approach. We are now investigating a wide range of applications involving speed, area and power consumption trade-offs.

References

[1] CoWare LISATek Tools. http://www.coware.com/.
[2] Tensilica. http://www.tensilica.com/.
[3] Altera Corp. http://www.altera.com/.
[4] MIPS CorExtend. http://www.mips.com/.
[5] IBM PowerPC. http://www.ibm.com/
[6] M. Jain et al., "ASIP Design Methodologies: Survey and Issues," Proceedings of the 14 International Conference on VLSI Design, 2001, pp. 3-7, Jan. 2001.
[7] D. Fischer, J. Teich, M.Thies, and R.Weper, “Efficient architecture/compiler co-exploration for asips,” in Proc. Int. Conf. Compilers, Arch., Synth. Embedded Syst., 2002, pp.27–34.
[8] N. Clark, H. Zhong, and S. Mahlke, “Processor acceleration through automated instruction set customization,” in *Proc. 36th Annu. Int. Symp. Microarchitecture*, Dec. 2003, pp. 129–140.
[9] P. Yu and T. Mitra, “Scalable custom instructions identification for instruction set extensible processors,” in *Proc. Int. Conf. Compilers Architectures Synthesis Embedded Syst.*, Sep. 2004, pp. 69–78.
[10] K. Atasu, L. Pozzi, and P. Ienne, “Automatic application-specific instruction-set extensions under microarchitectural constraints,” in *Proc. 40th Des. Autom. Conf.*, Jun. 2003, pp. 256–261.
[11] L. Pozzi, K. Atasu, and P. Ienne, “Exact and approximate algorithms for the extension of embedded processor instruction sets,” *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 25, no. 4, pp. 1209–1229, Jul. 2006.
[12] P. Yu and T. Mitra, “Disjoint pattern enumeration for custom instruction identification,” in *Proc. 17th Int. Conf. Field-Programmable Logic Appl.*, Aug. 2007, pp. 273–278.
[13] P. Bonzini and L. Pozzi, “Polynomial-time subgraph enumeration for automated instruction set extension,” in *Proc. Des. Autom. Test Eur. Conf. Exhibition*, Apr. 2007, pp. 1331–1336.
[14] X. Chen, D. L. Maskell, and Y. Sun, “Fast identification of custom instructions for extensible processors,” *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 26, no. 2, pp. 359–368, Feb. 2007.
[15] N.T. Clark, H. Zhong, S.A. Mahlke, “Automated custom instruction generation for domain-specific processor acceleration,” IEEE Transactions on Computers, Vol. 54, Issue. 10, p1258-1270, Oct. 2005.
[16] P. Ienne, L. Pozzi, and M. Vuletic, “On the limits of processor specialization by mapping dataflow sections on ad-hoc functional units,” Comput. Sci. Dept., Swiss Federal Inst. Technol. Lausanne, Lausanne, Switzerland, Tech. Rep. 01/376, 2001.
[17] F. Sun, S. Ravi, A. Raghunathan, and N. K. Jha, “Synthesis of custom processors based on extensible platforms,” in *Proc. Int. Conf. Comput.-Aided Des.*, 2002, pp. 256–261.
[18] J. Cong, G. Han, Z. Zhang, “Architecture and Compiler Optimizations for Data Bandwidth Improvement in Configurable Processors,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, Vol. 14, no. 9, pp. 986 – 997, 2006.
[19] Pozzi L. Pozzi and P. Ienne. Exploiting pipelining to relax register file port constraints of instruction-set extensions. In *CASES 2005*, San Francisco, CA, Sept. 2005.
[20] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "MiBench: A free, commercially representative embedded benchmark suite," Proc. IEEE 4th Ann. Workshop Workload Characterization (WWC 01), Dec. 2001, pp. 3-14.
[21] MPEG Audio Decoder. http://www.underbit.com/products/mad/.