社団法人 情報処理学会 研究報告
IPSJ SIG Technical Report

2004－SLDM－115 (9)
2004／5／28

例外検出用動的再構成可能なコプロセッサ

伴野 充[†] 中西 正樹[†] 山下 茂[†] 渡邉 勝正[†]

† 奈良先端科学技術大学院大学 情報科学研究科
〒 630－0101 奈良県生駒市高山町8916番地の5
E-mail: †{mitsu-to,m-naka,ger,watanabe}@is.naist.jp

あらまし 近年，リコンフィギャラブルコンピューティングは新しい計算パラダイムとして注目を浴びている．この計算パラダイムを用いる計算機は汎用プロセッサの柔軟性と専用システムのパフォーマンスを兼ね備えている．大抵の場合，アプリケーションプログラム中における処理の重たい部位をリコンフィギャラブルロジックにて処理することでシステムのパフォーマンス向上を図る．本論文では，リコンフィギャラブルコンピューティングにおける新しい活用法である例外検出への応用を提案する．我々の方法では，リコンフィギャラブルロジックは処理の重たいタスクではなく，比較的軽いが頻繁に処理が発生するタスクに用いられる．

キーワード リコンフィギャラブルコンピューティング，例外検出，動的再構成，コプロセッサ

Dynamically Reconfigurable Coprocessor for Exception Detection

Mitsuru TOMONO[†], Masaki NAKANISHI[†], Shigeru YAMASHITA[†], and Katsumasa WATANABE[†]

† Graduate School of Information Science, Nara Institute of Science and Technology
Takayama-cho 8916-5, Ikoma-shi, NARA, 630-0101, Japan
E-mail: †{mitsu-to,m-naka,ger,watanabe}@is.naist.jp

Abstract Recently, reconfigurable computing comes under the spotlight as the new computing paradigm. The machine employing this paradigm has the combination of the flexibility of the general purpose processor and the performance of the dedicated system. In most cases, compute-intensive tasks of application program are loaded to reconfigurable logics to increase the performance of the system. However, in this paper, we propose a new area for reconfigurable computing, which is *exception detection*. In our method, reconfigurable computing is exploited **not for** compute-intensive tasks **but for** highly frequently invoked (but relatively light) tasks.

Key words Exception Detection, Reconfigurable Computing, Dynamic Reconfiguration, Coprocessor

1. Introduction

During the past decade, reconfigurable computing [1], [2], [3], [4], [5], [6], has been researched extensively as a new method to improve processor performance. In particular, reconfigurable hardware coexisting with a core processor can be considered as a good candidate for speeding up processor performance. Such a hybrid processor demonstrates its effectiveness in multimedia applications, data encryption, signal processing, communications, and so on. The main idea in improving performance of such a solution is that reconfigurable logics process compute-intensive tasks of target application in lieu of general-purpose processor.

We investigate *another* potential area for reconfigurable computing in this paper, in which reconfigurable computing is exploited **not for** compute-intensive tasks **but for** highly frequently invoked (but relatively light) tasks. Namely, we propose to use a reconfigurable hardware for *exception detection* that has two main features: (1) We need to perform relatively *light* tasks (e.g., just check whether some variables are zero or not.), and (2) the same kind of tasks are invoked in highly parallel at the same time. In other words, our study might reveal a new application area for reconfigurable computing.

Exception handling is the essential mechanism to construct a robust and reliable software in the continuously expanding

computer system at the present day. As the system becomes enlarged, the scale and complexity of software is also increasing. Therefore, the software that performs continuously and stably has been desired. One of the promising ways to design such software with high availability is considered to handle all the possible exception errors by user program. This can be done by explicit use of a try and catch statement in JAVA or C++ language. By using a programming paradigm called Active Software [7], we can write exception handlings even in rather implicit way. In either way, a robust program should have a lot of exception handling routines in it, which essentially means that **many** error detections should **always** be performed throughout the program. As programs become larger and larger, this need of exception handling becomes larger and larger, and the overhead of the error detections will be surely a big problem in the near future. Therefore, it becomes important for the microprocessor to have the hardware to handle this operation without disturbing the normal operation.

To focus attention on operations of exception detection, we exploit the following three features of exception detection. (1) Since the required process of exception detection changes according to the situation, dynamic reconfigurable hardware is well-suited to meet the demands. (2) The hardware needs the capability to read register file of the main processor in parallel, to monitor the internal state of the main processor. (3) Highly parallel processing mechanism for exception detection is indispensable since the process must be carried out to all values of register file simultaneously.

In existing researches [8], [9], many dynamically reconfigurable architectures that have the capability to implement above-mentioned mechanism are proposed such as Chameleon [10], Chimaera [11], [12], [13], DISC [14], [15], DPGA [16], [17], Garp [18], [19], OneChip [20], PRISC [21], REMARK [22], [23], and Nano Processor [24]. As mentioned earlier, these reconfigurable architectures assume that some compute-intensive tasks are loaded to reconfigurable parts; data transfer from the main processor to reconfigurable parts should not be highly paralleled, and the reconfigurable parts should not return the response in very short time, i.e., few clock cycles. However, for our application, the following features are desired: highly parallel inputs of data and outputs of results. For example, although Chimaera has the capability to read its shadow registers in nine parallel, it is not well suited with respect to the feature mentioned above. Garp shares data cache with host processor and can read data through five 32bit buses, but, in terms of parallelism of data read, its capability is not sufficient. REMARK's reconfigurable blocks consist of 8 × 8 array of nano processors and a global control unit. However, its global control unit can transfer data to only 8 nano processors, so it is insufficient. The other architectures also do not have the enough capability to read register file of its main processor.

Therefore, any of them are not considered as suitable for exception detection hardware. For that reason, we propose well-suited reconfigurable architecture for exception detection.

The rest of this paper is organized as follows. In Section 2, we explain our exception handling model. Section 3 describes the architecture we propose. In Section 4, we show the evaluation of our architecture. Section 5 concludes the paper.

2. Exception Handling Model

Exception handling means the operation that is performed when unpredictable events such as arithmetic overflow, zero divide, use of undefined instructions, and requests from input or output devices take place during execution of program. Although the term 'exception' is used almost interchangeably with interrupt in general, we differentiate exception from interrupt. Exception means unpredictable events caused by internal factors and interrupt means unpredictable events caused by external factors.

In most cases, if some sorts of exception occur in a system, proper handling is executed in compliance with the type of exception. In our scheme, we associate exceptions with conditional expressions that include values of the processor's registers. Then we assume that an exception occurs when the corresponding conditional expression becomes true. We also assume that we know possible exceptions and their corresponding conditional expressions in advance. Given conditional expressions, the coprocessor evaluates them, and answers which are true. As stated in Section 1, to increase the availability and safety of software, many of the exception detections always must be done with the above method.

Possible computational exceptions vary according to application program. Therefore, implementation of conditional expressions should be changed depending on the situation. For that purpose, it is preferable that the system is able to reconfigure implementation of conditional expressions.

The operation of exception detection is considered relatively simple in a lot of cases. Hence, we consider the following three types of operations as conditional expressions. Of course, other types of operations can be easily implemented in our processor.

Type 1: $R1 \ op_{CMP} \ C1$

This is the most basic type of operation, the comparison of a value of register with the pre-established constant number.

Type 2: $(R1 \ op_{ALU} \ R2) \ op_{CMP} \ C1$

This type is the comparison of sum or difference of two values

with the pre-established constant number.

Type 3: $((R1\ op_{ALU}\ R2)\ op_{CMP}\ X)$ &&

$((R3\ op_{ALU}\ R4)\ op_{CMP}\ C2)$

This is the logical multiplication or logical addition of the Type 2.

Here, $R1$, $R2$, $R3$, and $R4$ are the values of some of the registers and $C1$ and $C2$ are the constant numbers. op_{CMP} is a comparison operator which is $>$, $<$, $>=$, $<=$, $==$, or $!=$. op_{ALU} is an arithmetic operator which is $+$ or $-$. The function of op_{CMP} and op_{ALU} is reconfigured per exception detection.

Considering all the above factors, our exception handling model is as follows.

(1) The exception detection hardware monitors register file of the main processor.

(2) At a given point in time, the main processor reconfigure the function of the coprocessor and it evaluates the conditional expressions for exception detection.

(3) If some sort of exception is detected, according to the detected exception, ID that corresponds to the detected exception is transferred to the main processor.

(4) When the main processor receives the ID, it interrupts the active program and processes exception handling.

Through the above-mentioned process, exception detection is performed.

In the following section, we describe the architecture to process exception detection in detail.

3. Hardware Exception Detection with a Dynamically Reconfigurable Coprocessor

3.1 Overview of Exception Detection

Figure 1 shows a block diagram of a processor into which **reconfigurable exception detection coprocessor (REDC)** is integrated. This hardware consists of the main processor, the **conditional expression evaluation module (CEEM)**, and the **ID handling module (IDHM)**. The main processor is a central processing element that performs application programs. The REDC is an exception detection unit. It evaluates conditional expressions and if the condition becomes true, it issues corresponding ID. IDHM is a controller. It transfers IDs to the main processor.

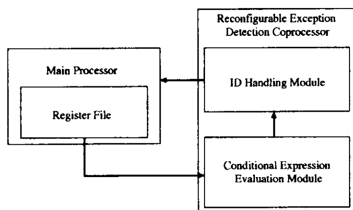

Fig. 1 Block Diagram

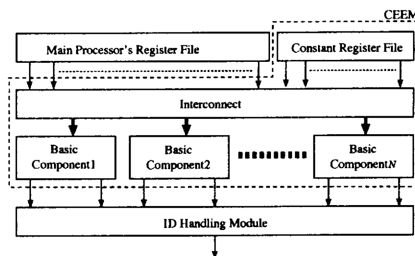

Fig. 2 Conditional Expression Evaluation Module

3.2 Reconfigurable Exception Detection coprocessor

The REDC consists of the following components: the CEEM for evaluation of conditional expressions and the IDHM for handling IDs issued by the CEEM.

The CEEM includes **basic components (BC)**, interconnect, and constant register file as shown in Figure 2. BC is a fundamental computational unit of the module. It evaluates conditional expressions in reference to main processor's register file. If a conditional expression is true, the BC sets 1 to the corresponding flag register. A constant register contains a constant number used in evaluation of conditional expressions. A BC can reconfigure its components' functions and inputs' values. Configuration data is stored in multicontext registers and one of them is selected with processor's context switching to decide the configuration of a BC.

The BC must execute three types of operations efficiently. Some types of the BCs are shown in Figure 3. In Figure 3, ALUs execute op_{ALU}, comparators execute op_{CMP}, and a zero or one comparator can judge whether an input value is zero or one. Although component 1_a can only evaluate a part of Type 1 operations, it is cost and area effective. Component 1_b can evaluate all of Type 1 operations. Component 2 can evaluate all of Type 1 and 2 operations. Component 3 can evaluate all of Type 1, 2, and 3 operations. For example, the component 1_a is sufficient in order to detect zero divide and we need the component 1_b to detect array index error. If we use the component 3 to detect such exceptions, the utilization of the hardware might become low. In practical exception detection, all types of the operations might exist at an irregular rate. Therefore, we need to consider the appropriate configuration of the BCs to detect various type exception detections efficiently. In the next paragraph, we show an example of the BC as a prototype for evaluation.

The example of the BC mainly consists of two arithmetic logical units, two comparators, and multiple inputs from reg-

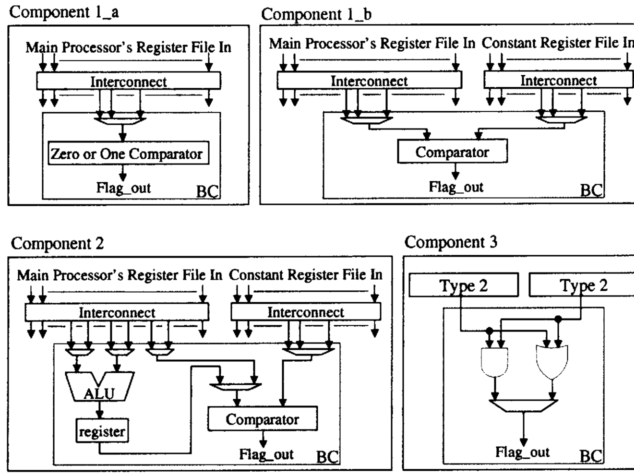

Fig. 3 Some Types of Basic Component

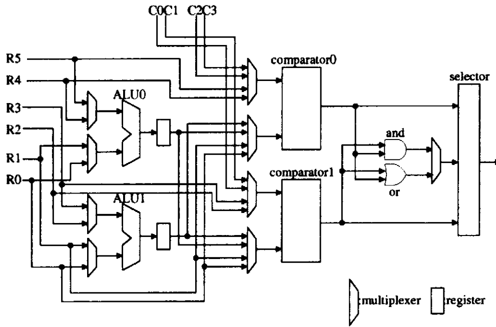

Fig. 4 An Example of Basic Component

ister file and constant register file. It can evaluate two conditional expressions of Type 1 or Type 2 simultaneously. Combining the two Type 2 expressions, it can evaluate a Type 3 expression. R0 through R5 are inputs of register file's values and C0 through C3 are inputs of constant register file's value. The flexibility of inputs of the BC is as follows.

BC_i can access to R_{i-2} *to* R_{i+3} and C_{i-1} *to* C_{i+2}

Here, i is the index of BCs. This BC can access to six registers and four constant registers. If a BC can access to all of register file's values and constant register file's values, interconnection routing and multiplexers to select data occupy much of the chip area. Therefore, if the data linking with each other are put together closely, we think the above flexibility is sufficient.

In the following example, we explain the flow of exception detection process. Suppose that at a certain point of the program, we need to examine whether variables exist in a certain range. Also suppose that at another point of the program, we need to examine whether variables are equal to zero. These conditions are described as follows.

Condition1

Exception ID	Conditional Expression	Operation Type
1	$X1 + X2 > 70$	Type 2
2	$X3 < 20$	Type 1
3	$X4 < -5 \&\& X4 > 5$	Type 3

Condition2

Exception ID	Conditional Expression	Operation Type
1	$X1 == 0$	Type 1
2	$X2 == 0 \&\& X3 == 0$	Type 3
3	$X4 == 0$	Type 1

Here, $X1$, $X2$, $X3$, $X4$ are the values of register $R1$, $R2$, $R3$, $R4$, respectively. Let Condition1 be assigned to context1 and Condition2 be assigned to context2. At the time to evaluate Condition1, the main processor switches the context number to 1 and makes the coprocessor to evaluate Condition1. If X3 is 10, BC_2 sets 1 to the corresponding flag register, and then, the IDHM transfer '2' as the detected exception ID to the main processor. In a similar manner, the coprocessor evaluates Condition2 and if any of the condition expressions is true, the corresponding ID is issued.

The IDHM fetches issued IDs to a queue, and then transfer them to the main processor one by one. To fetch issued IDs, the IDHM propagates the IDs on tree-formed data path, reducing the number of steps needed to fetch IDs to log#BC rather than #BC, where #BC is the number of BCs. Figure 5 shows the architecture of the IDHM. It consists of flag

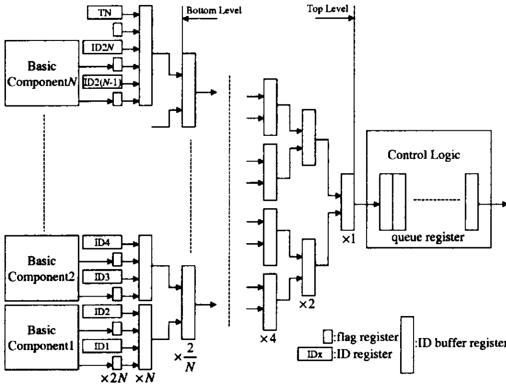

Fig. 5 ID Handling Module

registers, ID registers, ID buffer registers and the control logic block. Flag registers store the flag bit to which a BC outputs. ID registers are constant registers that provide IDs for exception handling. ID buffer registers are buffers for ID and flag bit. The control logic block consists of queue registers and simple control circuit. The queue registers are also buffers for ID and flag bit, and the control circuit controls the transfer of IDs from the queue registers to the main processor.

The structure of IDHM is the form of binary tree. Data are propagated from leaves to the root. It transfers IDs to the main processor according to the following step. (1) BCs set the results to the corresponding flag registers in parallel. (2) ID buffer registers of the bottom level in the binary tree fetch the value of the flag register and the ID register if the value of the flag register is 1. We consider a pair of an ID and the corresponding flag as a data unit. A buffer register fetches a pair of an ID and the value of flag register from one of the two leaves. If both of the flag register values are 1, the pair that has a smaller ID is given priority. (3) The pairs in ID buffer registers of each level are transferred to the next level ID buffer registers one after another. (4) Finally, the values go into the queue registers in the control logic block. (5) The main processor receives IDs from the queue registers and executes exception handling. This specific architecture of IDHM enables fast ID transfer.

4. Performance Evaluation

In this section, we present the performance evaluation of the REDC.

To evaluate the performance, we compare the execution time of conditional expression evaluation between the REDC and software implementation. The program used consists of the code that evaluates the 64 Type 2 expressions and, if the conditional expressions are true, it sets 1 to the corresponding flag. The environment of software implementation consists of intel Xeon processor running at 2.4 GHz and RedHat Linux 9 OS.

Table 1 Performance Evaluation of the REDC

	50MHz	100MHz	150MHz	200MHz
16 parallel	80nsec	40nsec	26.8nsec	20nsec
32 parallel	40nsec	20nsec	13.4nsec	10nsec

The time taken by the software implementation to execute the evaluation program is about 90nsec. On the other hand, considering the BCs can evaluate two Type 2 operations simultaneously in two clock cycles, the execution time of the REDC is shown in Table 1 at various parallelism and frequency.

From Table 1, the REDC which consists of 16 BCs at 50MHz can execute the program in almost the same time as 2.4 GHz Xeon processor. The REDC that consists of 32 BCs is 9 times faster than the software implementation if its clock frequency is 200MHz. The clock frequency of the REDC (200MHz) would be a reasonable assumption since it is less than one-tenth of the clock frequency of the processor. Therefore, the REDC can reduce the overhead of exception detection to a large degree.

To estimate this hardware, we have designed the REDC chip using verilog-HDL and implemented via VDEC Rohm LSI with 3.6mm × 3.6mm size, 0.35μm process, 2 PolySi layers and 3 metal layers. According to the result of post-layout simulation, the chip works at 50MHz. We expect that this performance becomes better if we use a dedicated library and manual layout.

5. Future Works

Future works include the following issues.

- A practical method for applying exception handling model to application software

To utilize this hardware in practical use, we need a framework to find potential exceptions, an automatic application code analyzer to do it, and a compiler to produce the corresponding code to the REDC and a main processor.

- Configuration of the BC

Although the prototype BC consists of two ALUs and two comparators, not all conditional expressions use these functional units and, in practical use, this configuration is considered to be redundant. Therefore, we do research on appropriate congifuration of the BC by combining some types of the BCs we proposed in Section 3.2.

- Flexibility of the interconnect

The flexibility of the interconnect decides the accessibility of data between the register file and BCs. The excessive flexibility leads to hardware overhead. Hence, we make a study

of suitable interconnect to feed data to BCs sufficiently.

6. Conclusions

In this paper, we have proposed a new reconfigurable coprocessor architecture that is specialized in exception detection. Through the use of the dynamic reconfiguration mechanism, it can detect exception depending on the changing situation. Therefore, this reconfigurable coprocessor can support the continuous and stable operations of software. As this architecture demonstrated, in the days to come, application of reconfigurable hardware might become interesting and important matters.

Acknowledgement

The VLSI chip in this study has been fabricated in the chip fabrication program of VLSI Design and Education Center (VDEC), the University of Tokyo in collaboration with Rohm Corporation and Toppan Printing Corporation.

This work is supported by MEXT.KAKENHI ((C)(2)15500023) and Okawa Foundation Research Grant.

References

[1] W. Mangione-Smith and B. Hutchings. Reconfigurable Architectures: The Road Ahead. *In Reconfigurable Architectures Workshop*, pp. 81–96, Geneva, Switzerland, 1997.

[2] J. Villasenor and W. Mangione-Smith. Configurable Computing. *Scientific American*, pp. 55–59, 1997.

[3] W. Mangione-Smith et al. Seeking Solutions in Configurable Computing. *IEEE Computer*, 30(12):38–43, 1997.

[4] D.A. Buell and K.L. Pocek. Custom Computing Machines: An Introduction. *Journal of Supercomputing*, 9(3):219–230, 1995.

[5] B. Kastrup, J. vanMeerbergen, and K. Nowak. Seeking (the right) Problems for the Solutions of Reconfigurable Computing. *In Proc. 9th Intl. Workshop FPL' 99*, pp. 520–525, Glasgow, Scotland, 1999.

[6] A. DeHon. Reconfigurable Architectures for General-Purpose Computing. A. I. 1586, Massachusetts Institute of Technology, Cambridge, Massachusetts, 1996.

[7] Robert Laddaga. Active software. *Proceedings of the first international workshop on Self-adaptive software*, pp. 11–26, Oxford, United Kingdom.2000.

[8] B. Radunović and V. Milutinović. A Survey of Reconfigurable Computing Architectures. *In Proc. 8th Intl. Workshop FPL' 98*, pp. 376–385, Tallin, Estonia, 1998.

[9] K. Compton and S. Hauck. Reconfigurable Computing: A Survey of Systems and Software. *ACM Computing Surveys*, 34(2):171–210, 2002

[10] Bill Salefski and Levent Caglar. Re-Configurable Computing in Wireless. *In Proc. 38th Design Automation Conference*, pp. 18–22. Las Vegas. 2001.

[11] Scott Huack, Thomas W. Fry, Matthew M. Hosler, and Jeffrey P. Kao. The Chimaera Reconfigurable Functional Unit. *IEEE Symposium on FPGAs for Custom Computing Machines*, 1997.

[12] Z. A. Ye, A. Moshovos, S. Hauck, P. Banerjee. CHIMAERA: A High-Performance Architecture with a Tightly-Coupled Reconfigurable Functional Unit. *International Symposium on Computer Architecture*, pp. 225–235, 2000.

[13] Z. A. Ye, A. Moshovos, S. Hauck, N. Shenoy, P. Banerjee. CHIMAERA: Integrating a Reconfigurable Functional Unit into a High-Performance, Dynamically-Scheduled Superscalar Processor. *http://www.ee.washington.edu/people/faculty/hauck/publications/YeTVLSI.pdf*

[14] M. Wirthlin and B. Hutchings. A Dynamic Instruction Set Computer. *In Proc. 3rd IEEE Symp. on FCCMs*, pp. 99–109, Napa Valley, California, 1995.

[15] M. J. Wirthlin and B. L. Hutchings. DISC: The Dynamic Instruction Set Computer. *in Field Programmable Gate Arrays (FPGAs) for Fast Board Development and Reconfigurable Computing*, pp. 92–103, Philadephia, PA, USA, Oct. 1995.

[16] A. Dehon. DPGA-coupled microprocessors: Commodity ICs for th early 21st century. *In Proc. the IEEE Workshop on FPGAs for Custom Computing Machines*, pp. 31–39. 1994.

[17] M. Bolotski, A. DeHon, T. F. Knight Jr. Unifying FPGAs and SIMD Arrays. *In Proc. 2nd International Workshop on Field-Programmable Gate Arrays*, Berkeley, California. pp. 1–10. 1994.

[18] J. R. Hauser and J. Wawrzynek. GARP: A MIPS processor with a reconfigurable coprocessor. *In Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 12–1, Apr. 1997.

[19] T. Callahan, R. Hauser and J. Wawrzynek. The GARP Architecture and C Compiler. *IProc. IEEE Computer*, 33(4), 6269, April 2000.

[20] R. Wittig and P. Chow. OneChip: An FPGA Processor With Reconfigurable Logic. *In Proc. 4th IEEE Symp. on FCCMs*, pp. 126-135, Napa Valley, California, 1996.

[21] R. Razdan and M. D. Smith. Highperformance Microarchitectures with Hardware programmable Functional Units. *In Proc. 27th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 172–80, November 1994.

[22] T. Miyamori and K. Olukotun. A Quantitative Analysis of Reconfigurable Coprocessors for Multimedia Applications. *In Proc. 6th IEEE Symp. on FCCMs*, pp. 2–11, Napa Valley, California, 1998.

[23] T. Miyamori and K. Olukotun. REMARC: Reconfigurable multimedia array coprocessor. *Proc. ACM/SIGDA FPGA '98*, Monterey, 1998.

[24] M. J. Wirthlin, B. L. Huthcings and K. L. Gilson. The Nano Processor: a Low Resource Reconfigurable Processor. *In Proc. the IEEE Workshop on FPGAs for Custom Computing Machines*, Napa CA. 1994.