Worst-case analysis of discrete systems

Felice Balarin
Cadence Berkeley Laboratories

Abstract

We propose a methodology for worst-case analysis of systems with discrete observable signals. The methodology can be used to verify different properties of systems such as power consumption, timing performance or resource utilization. We also propose an application of the methodology to timing analysis of embedded systems implemented on a single processor. The analysis provides a bound on the response time of such systems. It is typically very efficient, because it does not *require a state space search.*

1 Introduction

System verification is hard because system responses need to be checked for all legal behaviors of the environment. Typically, there are infinitely many such behaviors. Even when the problem can be reduced to enumerating finitely many internal system states, their number is usually prohibitive. Using abstractions and implicit state enumeration can simplify the problem, but complete verification is at best at (and often beyond) the limit of existing computers.

An alternative approach is the worst-case analysis, where the system response is analyzed only for the most demanding behaviors of the environment. Worst-case analysis is a well known engineering method, but so far it has been used ad-hoc, with separate techniques for specific system properties.

In this paper, we propose a general methodology for worst-case analysis of systems with discrete observable signals. It can be used to verify different properties of systems such as power consumption, timing performance, or resource utilization. In addition to the general methodology, we also propose its application to timing analysis of embedded systems implemented on a single processor. Timing analysis of embedded software system can be divided into two subproblems: local and global. The local subproblem is to determine processing time requirements for a piece of code implementing a single component of the system. The global subproblem is to determine response time of the system, given processing time requirements of system components, and taking into account that response to some requests may be delayed by responses to other requests. We address only the global subproblem.

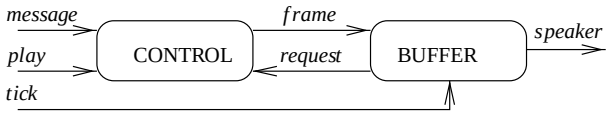
Related Work Our approach is based on the analysis of a system abstraction. Many researchers have suggested using abstractions to simplify formal verification of systems (e.g. [7, 6, 9]). In these approaches it is shown that abstraction preserves some properties of systems, so to verify a property of the detailed system, it is enough to verify it for the abstract one. In contrast, we propose an analysis of the abstract system and show how that analysis relates to the worst-case behavior of the detailed system. Thus, our work can be seen as an instance in the abstract interpretation framework [8]. It is important to note that while proving the properties of abstract systems typically requires searching their state spaces, our analysis does not.

The original motivation for our work was global timing analysis of embedded software. This problem has been addressed before, starting with Liu and Layland [10]. They give an exact solution, but only for a very restricted model. To fit a realistic system into this model, many conservative simplifications are required. The restrictions on the model have been somewhat relaxed later [2, 3], however several significant limitations are present in all the previous approaches, but not in the approach presented here:

- The processing time requirements of a component were assumed to be constant. In contrast, we allow them to be a function of the inputs and internal states.
- An execution of a component is assumed to cause executions of all of its successors, while in reality a component may be enabling only some of them, depending on the inputs and internal states.
- Previous approaches were restricted to acyclic system graphs, i.e. to systems for which there exists a well defined unidirectional information flow.
- Previous approaches were applicable only to systems with static priority scheduling.

In theory, a general and exact solution could be obtained by modeling the system in sufficiently expressive formalism

0-7803-5832-X /99/$10.00




```
module CONTROL { frameType frames[5]; integer last:=0;
1 if( present( message)){
2   frames := value( message );
3   last := size( value( message ));}
4 if((present( play) || present( request)) && last > 0){
5   emit frame( frames[ last−−]);}}

module BUFFER { sampleType samples[50]; integer last:=0;
6 if( present( frame)) {
7   samples := value( frame);
8   last := 50;}
9 if( present( tick) && last > 0)) {
10  emit speaker( samples[ last−−]);
11  if(last = 20)) {
12    emit request();}}}
```

Figure 1. Voice mail pager.

(i.e. timed automata [1]). However, such an approach is orders of magnitudes slower than the worst-case analysis, and practically impossible for realistic systems [5].

Voice mail pager Throughout this paper we use as an example a voice mail pager shown in Figure 1. To facilitate future references, we briefly explain its behavior. The pager receives messages from the environment. Each *message* consists of up to five frames, and each frame contains fifty samples. The CONTROL module stores messages internally (in variable *frames*) and initiates playing of the most recent message (by generating the *frame* event), when the user requires so (by generating the *play* event). The CONTROL module also generates a *frame* if some are available and the BUFFER module makes a *request*. The BUFFER module starts playing the message after it receives the initial *frame* from CONTROL. A message is played by sending to the *speaker* one sample per every *tick* of the real-time clock. When there are fewer than 20 samples left to play BUFFER sends a *request* for the next frame to CONTROL.

Overview Our approach is sketched in Figure 2. The worst-case analysis is performed on a user-provided abstraction of the system (lower part of Figure 2), rather than on the system itself. In the abstraction, signals are replaced with their *signatures*. Abstracting the signals leads to some information loss, but signatures should retain enough information to estimate activity of system components. For example, the signature of inputs to CONTROL module should contain enough information to estimate (i) how much activity of the *frame* signal CONTROL will generate, and (ii), how much time, energy, memory bandwidth or any other resource of interest CONTROL needs to process its inputs. These estimates are computed by component abstractions (F_{CONTROL} and F_{BUFFER} in Figure 2).

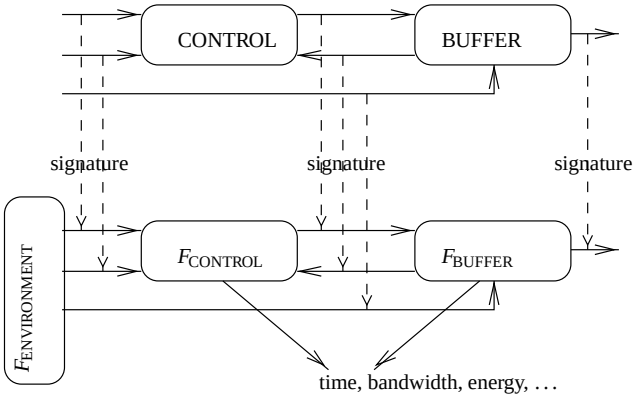


Figure 2. Worst-case analysis overview.

The main result of this paper is that by analyzing the abstraction, it is possible to find the worst-case signature, i.e. a signature that is worse than the signature of *any* execution of the system. The focus of this paper is the mathematical foundation of the worst-case analysis. We formalize the analysis and give precise conditions the signature and component abstractions must satisfy for the analysis to be valid. Meeting these conditions guarantees that the result of analysis is indeed worse then the signature of any execution, but there is no guarantee that this bound is useful. The computed bound may be overly conservative, or even trivial (e.g. "the worst-case energy is not more than infinity"). The quality of the bound depends on the information preserved in the signature, and on the "tightness" of the abstraction of system components. How to chose an appropriate signature, and how to construct a tight system abstraction for that signature are both important topics, but they are beyond the scope of this paper. Here, we only present a case study to indicate that for at least one realistic design it is reasonably easy to construct signatures and abstractions that lead to almost exact performance bound.

The rest of this paper is organized as follows. In Section 2 we introduce all relevant definitions. The general worst-case analysis is proposed in Section 3. Its application to response time analysis of embedded software systems is proposed in Section 4. A case study is presented in Section 5. We give some final remarks and indications of the future work in Section 6.

2 Systems, signatures and abstractions

First, we introduce a very simple and general formal notion of systems. A *system* is a set of executions. An *execution x* of length $len(x)$ is a mapping from the interval of real numbers $[0, len(x)]$ to some set of *signal values* such that it is *piecewise-constant*, *right-continuous* and has only *finitely many discontinuities*, i.e there must exist finitely

many points $t_0, \ldots, t_n$ such that $0 = t_0 < \ldots < t_n = len(x)$, and x is constant in $[t_{i-1}, t_i)$ for every $i = 1, \ldots, n$.

We require that all executions in a system range over the same set of signal values. In fact, we assume that all executions mentioned in this paper range over the same set, so we never need to specify it explicitly. The executions of a system need not be all of the same length. In fact, a more complete definition would require that a system contains execution of any real length, and that the set of executions is prefix closed, but these technical details are not necessary for our purposes.

For example, for the pager in Figure 1 signal values might be state variables and communication events. To satisfy the constraint that executions are right-continuous, we may assume that an event retains its value until the next event occurs on the same connection.

Given an execution x and real numbers u, t such that $0 \le u \le t \le len(x)$, we use $x[u,t]$ to denote an execution of length $t-u$ defined by: $x[u,t](v) = x(u+v)$. If, in addition, $u > 0$, we use $x[u,t]^-$ to denote $x[u-\varepsilon, t-\varepsilon]$, where $\varepsilon > 0$ is small enough that x is constant in $[u-\varepsilon, u)$ and $[t-\varepsilon, t)$. Since x has only finitely many discontinuities, such a choice of ε is always possible.

2.1 Signatures

In our approach, instead of manipulating executions directly, we use their abstractions called *signatures*. Formally, *signature* σ is a mapping from the set of executions to some set of *signature values* D_σ, for which some partial order $\le$ is defined, such that for all u, t satisfying $0 \le u \le t \le len(x)$:

1. $\sigma(x[u,t]) \le \sigma(x)$
2. $\sigma(x[u,t]) = \sigma(x)$ if $x(v) = x(u)$ for all $v \in [0,u]$ and $x(v) = x(t)$ for all $v \in [t, len(x)]$.

Intuitively, a signature represents a summary of activities in an execution. This intuition is consistent with the constraints we place on σ: a full execution should contain more activities than its portion (condition 1), and the activity summary should not change if an execution is extended with passive (i.e. constant) segments (condition 2).

For the pager in Figure 1 we consider the signature represented by the following vector of variables:

$$(pl, ms, tk, fr, rq, sp) \ .$$

For a given execution, we choose variable pl (*ms*, *tk*, fr, *rq*, *sp*) to hold the number of occurrences of the *play* (*message*, *tick*, $frame$, *request*, *speaker*, respectively) events in that execution. Figure 3 shows a component of an execution corresponding to CONTROL module and its signature.

By definition, signature values must be partially ordered. In this case, we use a natural extension of relation $\le$ (on reals) to vectors: two vector are related by $\le$ if and only if all their components are.

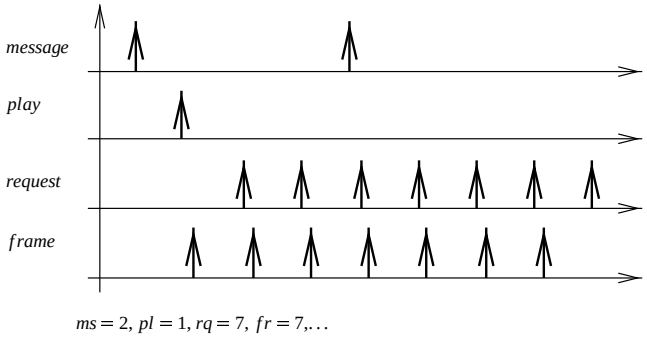


Figure 3. An execution and its signature.

2.2 Signature based abstractions

Signatures provide abstract representation of signals in the system. Next, we define the abstract representation of the system transition function.

Given some system S, signature σ and some function $F : D_\sigma \times \mathbb{R}^+ \mapsto D_\sigma$, we say that F is a σ-*abstraction* of S if F is:

- *monotone*: if $T_1 \le T_2$ and $s_1 \le s_2$ then $F(s_1, T_1) \le F(s_2, T_2)$, and
- *future-bounding*: for every execution $x \in S$ and every u, t: $\sigma(x[u,t]) \le F(\sigma(x[u,t]^-), t-u)$.

Consider, for example, the pager in Figure 1 and the signature σ discussed in the previous section. Its σ-abstraction consists of a separate function for each variable in the signature. The variables can be divided into three groups: those generated by the environment, those generated by the CONTROL, and those generated by the BUFFER module. The σ-abstraction of the environmental variables depend only on time and not on other signature variables. We use a very simple model, where the only constraint is a minimum time between two occurrences of the event. These times are 125, 625, and 5000 time units for *tick*, *message* and *play* events, respectively. Therefore, the maximum number of *tick* (*message*, *play*) events in an execution segment of length T is $\lfloor \frac{T}{125} \rfloor + 1$ ($\lfloor \frac{T}{625} \rfloor + 1$, $\lfloor \frac{T}{5000} \rfloor + 1$, respectively).

σ-abstractions of CONTROL and BUFFER variables are derived from the code in Figure 1. For example, BUFFER generates a *request* event 30 *tick* events after it receives a $frame$. Thus, in the interval containing fr $frame$ events and tk *tick* events, the BUFFER module can generate at most $\min(fr, \lfloor \frac{tk}{30} \rfloor)$ *request* events, except possibly an extra one at the beginning of the interval. Thus:

$$F_{rq} = \min(fr, \left\lfloor \frac{tk}{30} \right\rfloor) + 1 \ .$$

Other σ-abstractions are derived by similar reasoning [4].

3 Worst-case analysis

The problem of the worst-case analysis of a system S with a signature σ and σ-abstraction F is the following:

> For a given interval length T, find a signature value s such that $s \geq \sigma(x[t,t+T])$ for every execution $x \in S$, and every $t \in [0, len(x) - T]$.

In other words, s must be worse than the signature of any execution segment of length T. Such information can help answer many important questions in the design process. For example, if the signature contains information about bus requests, then the worst-case analysis indicates required average bus bandwidth for any period of time of length T. Similarly, if the signature contains information about energy required for an execution, then the worst-case analysis gives a bound on the average power for any interval of time of length T. The worst-case analysis can also be used to analyze timing performance of the system, as will be shown in Section 4.

There are two cornerstones of the worst-case analysis: partial ordering of signatures, and future-bounding property of σ-abstraction. Partial ordering of signatures enables us to rank executions and consider only "bad" ones. Intuitively, larger signatures correspond to executions with more activities, which usually represent more demanding cases for the system.

The future-bounding property of σ-abstractions is used to compute such a worst-case signature. We first establish that a candidate signature is worse than the signature of the initial segment of any execution, and then we use future-bounding property to show that it remains worse for any execution segment of length T. We can make this argument if the candidate signature is also a fix-point of the σ-abstraction. If the signature of some execution segment is less than a fix-point of the σ-abstraction, then the future-bounding property assures that the signature of the segment in the immediate future is also less than that fix-point. This reasoning is formalized by the following result:

Theorem 1 *Let σ and F be a signature and a σ-abstraction of some system S. If $x \in S$, $s \in D_\sigma$, and $T \geq 0$ are such that $s = F(s,T)$ and $\sigma(x[0,0]) \leq s$, then:*

$$\sigma(x[t,t+T]) \leq s \quad \forall t \in [0, len(x) - T] \ .$$

The proof proceeds by induction on constant segments of an execution [4]. Based on Theorem 1, the worst-case analysis proceeds through the following phase:

1. choose a signature σ and prove it meets conditions required by the definition,
2. choose a σ-abstraction F and verify that it is monotone and future bounding,
3. find a signature s such that $s = F(s,T)$ for a given T,
4. interpret the results.

Choosing appropriate signatures and σ-abstractions is an art that requires understanding both the system and the property to be analyzed. Checking properties of signatures and monotonicity of σ-abstractions is straightforward mathematical exercise that is independent of the system being analyzed. Checking future-bounding property is a typical verification problem: it amounts to checking whether a detailed "implementation" (in our case the system) satisfies a more abstract "specification" (σ-abstraction). Usual formal and informal verification approaches can thus be used. As our example indicates, σ-abstractions are often vectors of functions, one for each system component. This decomposition simplifies verification of the future-bounding property.

Solving the fix-point equation required by Theorem 1 is typically quite simple. In the next section, we will propose an iterative solution method for a slightly harder problem. The same method could be applied here.

The final step of the worst-case analysis is interpreting the computed worst-case signatures. If the signature abstraction is chosen carefully, then the worst-case signatures will contain enough information for the designer to determine bounds on many important quality measures of the design. We have mentioned three: bus utilization, power consumption, and timing performance, but we believe that other interesting properties may be analyzed as well.

4 Busy-period analysis

In this section we apply the worst-case analysis to bound response time of a software system implemented by a single processor. More precisely, we bound the *busy period*, i.e. longest period of time a processor can be busy. A bound on a busy period is also a bound on the response time, under the reasonable assumption that the processor cannot be idle if there are pending requests.

To formalize the notion of the processor being busy, we assume that predicate $Busy(x(t))$ is defined for every execution x and every time $t \in [0, len(x)]$. We say that some interval $[u,t]$ is a *busy period* of x if $Busy(x(v))$ holds for each $v \in [u,t]$. We say that busy period $[u,t]$ of x is *initialized* if there does not exist $v < u$ such that $[v,t]$ is a busy period of x.

To compute a bound on busy periods we need information about processing time requirements. We require that this information is provided by a *workload function*. The

processing time requirements depend on events in the environment. Therefore, the workload function should depend on an execution. However, to enable worst-case analysis, we only consider an abstraction which depends on a signature.

Formally, for a given system S and signature σ, we say that $R : D_\sigma \mapsto \mathbb{R}$ is a *workload function* if it is:

- *monotone*: if $s_1 \leq s_2$ then $R(s_1) \leq R(s_2)$, and
- *workload-bounding* for every execution $x \in S$, and every initialized busy period $[u,t]$ of x:

$$R(\sigma(x[u,t])) > t - u \ .$$

The workload-bounding property ensures that approximate processing time requirements given by a workload function is a strict upper bound on the actual processing time requirements which determines the busy period length.

For example, assume that the execution time of each executable line in Figure 1 is 10 time unit. Also, assume the signature discussed in Section 2.1. Then, a workload function for the pager might be:

$$\begin{aligned} R \quad = \quad & 20*(pl+rq+ms)+20*ms+10*fr+ \\ & 20*(fr+tk)+20*fr+20*sp+10*rq \ . \end{aligned} \quad (1)$$

The first line in (1) corresponds to the CONTROL module, while the second line corresponds to the BUFFER module. The term $20*ms$ in (1) is due to lines 2 and 3 in Figure 1, which will be executed only if a new *message* is received. Similarly, the number of executions of line 5 is the same the number of generated $frame$ events (hence the term $10*fr$), and so on.

The workload functions and σ-abstractions can be combined to bound the length of busy periods. Let T be the length of some initialized busy period. According to Theorem 1, its signature is bounded by s such that $s = F(s,T)$. It follows that the required processing in that busy period, and therefore also T, is bounded by $R(s)$. More formally:

Theorem 2 *Let σ, F, and R be a signature, a σ-abstraction, and a workload function of some system S. If $s \in D_\sigma$, and $T \geq 0$ are such that $s = F(s,T)$, $T = R(s)$, and $\sigma(x[0,0]) \leq s$ for all executions $x \in S$, then T is an upper bound on the length of busy periods for all executions $x \in S$.*

The proof is quite simple [4]. To apply Theorem 2, we need to solve fix-point equations $s = F(s,T)$, $T = R(s)$. We propose the following simple iterative algorithm:

1. let $T := 0$, let s be such that $s \geq x[0,0]$ for all executions x,
2. let $s := F(s,T)$, let $T := R(s)$,
3. repeat step 2 until convergence or until $T \geq T_{\mathrm{MAX}}$,

where T_{MAX} is a user-given bound. It is reasonable to ask for such a bound, because systems usually have some minimal performance requirements. If the bound is known to be higher than this minimal requirement, there is little point in wasting resources in determining exactly how unacceptable the performance is. The iteration will always terminate providing that R satisfies some reasonable assumptions [4].

5 Case study

We have applied the busy-period analysis to a voice-mail pager that is much more complex and realistic than the one presented in this paper. The design has a total of 13 modules, 4 of which are intended to model the environment. We have studied a case where 9 other modules where all implemented in software running on a single processor. The total size of the design was approximately 2500 lines of C code. The pager needs to service several periodic and aperiodic requests, The most frequent one of these repeats every $125\mu s$, hence the requirement that the maximum busy period be less than $125\mu s$.

The original simulation test-bench for the design tested the scenario where a single message was received and then played. To compare the results of the busy-period analysis to those of simulation, we have developed a σ-abstraction of the environment that is valid for that case (single message only). The longest busy-period observed in the simulation trace was $82\mu s$, while the the busy-period analysis provided an upper bound of $83\mu s$. These results differ by less than 2%, and they were both well within the $125\mu s$ requirement.

In the second experiment we have developed a general σ-abstraction of the environment that was no longer limited to a single message. In that case the busy-period analysis gives a bound of $148\mu s$ (violating the $125\mu s$ requirement). Careful analysis of the results indicated that this worst case can appear only if one message is received and then played, while another message is being played. Based on this information, we were able to construct a simulation trace which contains a $146\mu s$ busy period.

An interesting question is whether such a trace would be included in a more comprehensive randomized set of simulation vectors. The analysis of the worst-case trace indicated that to exercise such a behavior, the request to play the second message must come within a $66\mu s$ wide time window that occurs once in every $6250\mu s$. Assuming the uniform distribution for the probability of requests, this analysis indicates that the probability of exercising the worst case behavior is just over 1%, even if the required scenario (one message is received and played while another message is being played) is being tested.

In this case run times for the busy-period analysis were much shorter the the simulation run times ($20ms$ vs. 1

minute). However, this comparison is somewhat unfair, because the busy-period analysis was done by a piece of code written specifically for this example, while the reported simulation times are for a general discrete-event simulator. In general, we expect run times of two approaches to be comparable, because solving the fix-point equations in the busy-period analysis is equivalent to simulating the (abstracted) system up to (at most) time T_{MAX}.

6 Conclusions

We have proposed a methodology for worst-case analysis of systems with discrete signal values. We have also applied this methodology to response time analysis of reactive uni-processor systems.

Our method requires almost no assumptions about the scheduling algorithm used to control processor sharing among system components. However, if some information about the scheduling is known, it can be used to improve the accuracy of the analysis. For example, if a preemptive static priority scheduling is used, we can easily extend our analysis to p-busy-period analysis: to bound the interval of time a processor is continuously busy executing at priority level p or higher. Essentially, p-busy-period analysis is just a regular busy-period analysis, but for a modified system in which all components with priority less than p have been removed. A bound on p-busy period is a better bound on response time for requests at priority p.

Another way to use scheduling information is to improve σ-abstractions and workload functions. For example, if we know that the BUFFER module in Figure 1 is scheduled once per every $tick$ event, we can strengthen (1) by replacing the terms $20 * (fr + tk)$ and $20 * fr$ with $20 * tk$ and $20 * \max(fr, tk)$ respectively.

In the future, we plan to investigate whether σ-abstraction can be automated for some common system models (e.g. finite state machines). We also plan to use system invariants to improve accuracy of the analysis.

References

[1] R. Alur and D. L. Dill. Automata for modelling real-time systems. In M. Paterson, editor, *ICALP'90 Automata, languages, and programming: 17th international colloquium.* Springer-Verlag, 1990. LNCS vol. 443.

[2] N. C. Audsley, A. Burns, M. Richardson, K. W. Tindell, and A. J. Wellings. Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering Journal*, pages 284–292, Sept. 1993.

[3] F. Balarin. Priority assignment for embedded reactive real-time systems. In F. Mueller and A. Bestavros, editors, *Languages, Compilers, and Tools for Embedded Systems, ACM SIGPLAN Workshop LCTS'98 Montreal, Canada, June 1998*, pages 146–155. Springer, 1998.

[4] F. Balarin. Worst-case analysis of discrete systems. Technical report, Cadence Berkeley Laboratories, May 1999.

[5] F. Balarin, K. Petty, A. L. Sangiovanni-Vincentelli, and P. Varaiya. Formal verification of the PATHO real-time operating system. In *Proceedings of 33rd Conference on Decision and Control, CDC'94*, Dec. 1994.

[6] S. Bensalem, A. Boujjani, C. Loiseaux, and J. Sifakis. Property preserving simulations. In G. v. Bochmann and D. Probst, editors, *Proceedings of Computer Aided Verification : 4th International Workshop, CAV '92, Montreal, Canada, June 29-July 1, 1992.* Springer-Verlag, 1993. LNCS vol. 663.

[7] E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. In *Proc. Principles of Programming Languages*, Jan. 1992.

[8] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction of approximation of fixpoints. In *Proc. 4th Ann. ACM Symp. on Principles of Prog. Lang.* 1977.

[9] R. P. Kurshan. *Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach*. Princeton University Press, 1994.

[10] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-realtime environment. *Journal of the Association for Computing Machinery*, 20(1):46–61, January 1973.